Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática,
2013

Nuno Miguel Pereira Mogas da Silva

Sistema computacional para análise e redesenho de genes
Computational system for gene analysis and redesign

Nuno Miguel Pereira Mogas da Silva

Sistema computacional para análise e redesenho de genes
Computational system for gene analysis and redesign

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Dr. José Luís Oliveira (Professor Associado da Universidade de Aveiro e Investigador no IEETA) e da Dr.ª Gabriela Moura (Investigadora Auxiliar, Departamento de Biologia e CICECO, Universidade de Aveiro).

o júri / the jury

presidente / president

Armando José Formoso de Pinho
Professor Associado com Agregação da Universidade de Aveiro

vogais / examiners committee

José Luís Oliveira
Professor Associado da Universidade de Aveiro (orientador)

Rui Pedro Lopes
Professor Coordenador do Departamento de Informática e Comunicações do Instituto Politécnico de Bragança

agradecimentos / acknowledgements

Agradeço em primeiro lugar ao meu professor, e orientador, José Luís Oliveira, pelo acompanhamento, disponibilidade e ajuda.

À minha co-orientador Gabriela Moura, pela disponibilidade e ideias que apresentou e, principalmente, por todas as dúvidas esclarecidas relacionadas com genética.

Agradeço em especial ao Paulo Gaspar, membro do grupo de bioinformática do IEETA, pelo constante acompanhamento, disponibilidade, troca de ideias, orientação e ajuda durante todo este trabalho.

Agradeço a todo o grupo de bioinformática do IEETA.

Agradeço a todos os meus colegas e amigos de curso, pelo apoio e convivência ao longo destes anos.

Agradeço à Ana Rosa por todo incentivo e apoio momentos de maior desânimo.

Um agradecimento em especial aos meus pais, pela oportunidade que me deram de poder estar aqui, por todos os valores que me ensinaram e pelo suporte e apoio incondicional.

Resumo

A evolução das tecnologias permitiu ao homem explorar diversas áreas de forma mais eficiente e rápida. Uma das áreas onde a informática e a computação têm um grande impacto é a biologia, permitindo aos investigadores resolverem tarefas de forma mais eficiente, e em tempo útil, sem recorrer à experimentação prática em laboratório. Em biologia molecular, inúmeros métodos computacionais são usados, como por exemplo, na sequenciação e anotação de genomas e também em métodos de redesenho de genes.

Por sua vez, com o crescente poder computacional, procura-se realizar o maior número de tarefas no menor tempo possível, recorrendo-se para isso a diversas metodologias de otimização. Estas podem ser apresentadas de diversas formas, desde o recurso a mais memória e melhor desempenho do hardware bem como a algoritmos mais eficientes.

Esta tese procura avaliar de que forma distintos fatores associados às características de cada gene ajudam a explicar a evolução destes para o seu estado atual. Os padrões evolutivos potenciados por este estudo podem igualmente ser utilizados como motivos principais no redesenho de genes. Estas tarefas são computacionalmente dispendiosas e os tempos de execução elevados devido às muitas combinações de diferentes métodos que são realizadas. Para minorar este problema, esta tese apresenta também algumas soluções para otimizar os métodos de redesenho de genes, de forma a que estes obtenham os mesmos resultados num menor tempo possível.

Abstract

Technology evolution has allowed man to explore different areas more efficiently and quickly. One of the areas where informatics and computation have a major impact is biology, allowing researchers to solve tasks more efficiently, and on time, without resorting to laboratory experimentation. In the field of molecular biology, several computational methods are used, for instance, genome sequencing and annotation and also in gene redesign methods.

Meanwhile, with the growing computational power, there is a need to make the greatest number of tasks in the shortest time possible, using for it several optimization methodologies. Those can be presented in different forms, from the usage of more memory and better performance provided by the hardware as well as more efficient algorithms.

This thesis tries to explore how distinct factors related to each gene characteristics can aid to explain how each gene evolved to its current state. The evolutionary patterns enhanced by this study can also be used as the main reasons in gene redesign. These tasks are computationally expensive and their execution times are high due to the various combinations performed with different methods. To mitigate this problem, this thesis also presents some solutions to improve the gene redesign methods performance, in a way that they can achieve the same results in a shorter time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Every living being is built according to a genetic library, called genome, which is made of Deoxyribonucleic Acid (DNA). From the ancient times, genomes have suffered successive changes that lead to the evolution of the organisms. Some of those mutations are caused by errors during DNA duplication (replication), a crucial step so that a new copy of the genome can pass to the next generation. Other mutations can also be caused by the environment where the organism live. In other words, evolution is driven by mutations of the DNA.

Genes are nothing more than portions of an organism's genome, that carry the code used for building the molecular structures that are key for living organisms. Therefore, understanding how a gene originates is still a field that requires special attention from researchers. To do so, they often need to express those genes in laboratory, using a host species. Hence, several tools aid researchers investigating and better understanding the process of synthesizing proteins in host species in a quick and accurate faction, based on several known algorithms that redesign the DNA sequence to improve the protein yield and quality.

On the other hand, today's society want results faster and with better quality. In computer science this means that a program, must be efficient, accurate and fast to execute its goal. Studying and developing new algorithms and optimization techniques, that can reduce execution time, is a major challenge and a motivation for computer science enthusiasts.

This thesis explores two distinct fields: gene for heterologous expression optimization algorithms and algorithmic optimization. Using distinct gene design algorithms, this work tries to determine what factors influenced a gene to mutate into its current state. Also, this normally needs a huge computing time, since it explores millions of possibilities that can explain a gene's evolution. Also, optimization techniques should be applied to the known gene optimization methods in order to reduce their overall execution time. Furthermore, combining all this achievements into a single package should result in a dataset that can be used by researchers to explain what biological processes a gene went through until it reaches its current state. This also served as a motivational purpose for the development of this thesis.

1.2 Objectives

The set of informatics solutions for gene sequence redesign grows everyday. Also, there are an evergrowing spring of genetic digital data that needs to be evaluated to explain genes

evolution.

EuGene[1] a former software package for multivariate gene optimization for heterologous expression, already processes and analyzes some information about genes, using biological concepts like codon usage, GC content (guanine-cytosine content) or hidden stop codons. Hence, the goal of this thesis is not only to explore further approaches for gene sequence redesign and optimization for heterologous expression but also try to combine those methods to explain how a gene may have evolved to its current state. This project objectives included the following:

- Study and implement additional gene redesign algorithms into EuGene software, such as:
  - Rare Codons
  - Codon Correlation Effect
  - Ramp Effect
- Study and implement performance optimization techniques for the gene redesign process.
- Enhance the global optimization block used in EuGene (Simulated Annealing algorithm) to decrease its execution time.
- Study and implement an architecture that can combine all the gene redesign methods in order to find the best input parameters that conducted a gene to its present form.

From an engineering point of view, creating a system that explores all gene redesign methods in an exhaustive way is a tenacious job, being the reason why this thesis outline resulted in an exploratory work where once the results are achieved, they are stored for future investigation. Thus, this analysis should be performed once for every gene of a genome, and the results should be made available for later use by biology researchers. Also, a major goal of this thesis is to decrease every gene redesign method execution time as much as possible to decrease the computation time of each analysis.

1.3 Thesis outline

This thesis is organized in seven distinct chapters, being the remaining six briefly described bellow:

Chapter 2 presents some biological background needed to support this thesis goal. Here it is presented the basic principals regarding genetic information as well as some common biological concepts. This includes the process behind protein synthesis, what are synonymous codons and why they are important, and some gene redesign methods available in the literature. Moreover, the software that served as *core* for this thesis is presented showing its features and how they were explored.

[1] http://bioinformatics.ua.pt/EuGene/

Chapter 3 explains the main requirements and the architecture of the project. Here the user requirements are detailed, enhancing why this project is important for researchers. Also, from the engineering point of view, the functional and non-functional requirements are listed. This includes the architecture behind the software and how it will work and aid the researchers.

Chapter 4 shows the gene redesign methods performance and how they can be optimized. Here it is analyzed where the system overall performance can be improved. Moreover, it details every gene redesign method and presents some methodologies used to improve their global performance, reducing the time they take to achieve an optimal result.

Chapter 5 shows how the redesign methods were used to achieve this thesis goals. Also, it presents how gene redesign methods parameters were chosen, how random genes were generated and also how to calculate a similarity score between gene sequences. Moreover, the structure behind the data storage is presented and how it can be used by researchers.

Chapter 6 discusses the improvements achieved with the decisions taken in chapter 4. It presents all individual gene redesign analyzes, showing how faster the new plugins version are, when compared directly with their older version. A quick summary with the improvements achieved for each redesign method is also presented.

Chapter 7 shows the conclusions of this work. It explains how the optimizations were important, and a validation of how the redesign methods can be used for other purposes regarding EuGene. Finally this chapter also points out some research lines for future work.

Chapter 2

Genetics background and EuGene

2.1 Protein synthesis

When we think about genetics, most of the times we associate this concept with DNA (deoxyribonucleic acid). Nevertheless, genetics can be described as the science which studies the heredity, variation, molecular structure and function of genes in living organisms.

The DNA molecules are present in almost every cell of a person's body and the same is true in almost all other organisms. Hence, it is important to know what kind of information DNA carries. DNA properties allows it to function as a very efficient vehicle to store information [1]. DNA stores information, encoded as a sequence of four chemical bases: adenine (A), guanine (G), cytosine (C), and thymine (T). The order in which these bases appear determines the information available for building and maintaining an organism, similar to how we use the alphabet to build different combinations of letters in order to build words and sentences. Most DNA molecules are structured as a double-helix and the four nucleotides (bases) are located as shown in the figure 2.1:

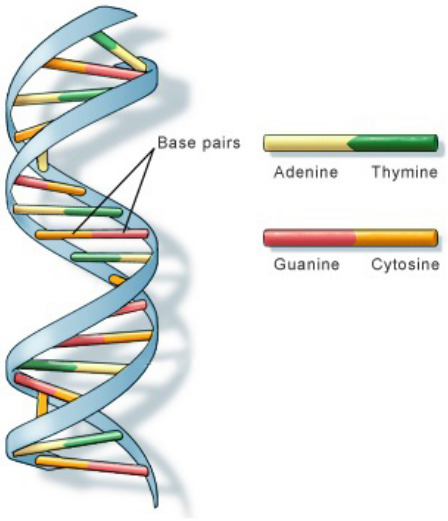


Figure 2.1: DNA molecules with double-helix form. All four different nucleotides are presented with different colors.

The DNA is a continuous sequence of nucleotides as stated before, although we can divide

it in smaller sequences of interest, called genes. This division is important because each gene can define an organism characteristic, for instance, in human DNA we can find genes that determine one's height, eye color, etc.

This genetic information is used through a process that copies the nucleotide sequence of the gene and produces another nucleic acid, similar to DNA, called RNA. The process of synthesizing RNA from DNA is called a transcription [2].

Although RNA can be seen as a copy of DNA, it differs in certain ways, for instance: while DNA is formed by a double-stranded chain, RNA is formed by a single-stranded chain; RNA has the bases Adenine (A), Uracil (U), Cytosine (C) and Guanine (G), i.e, the thymine base is replaced by Uracil.

A RNA transcript is composed by exons and introns. Exons are the nucleotide sequence that remains present in the final messenger RNA (mRNA) sequence, as presented in figure 2.2, while introns are the sequences that are discarded during mRNA maturation.

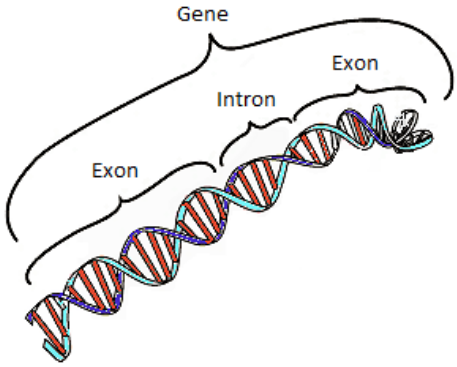


Figure 2.2: Portion of DNA: gene. DNA is transcripted to RNA and the resulting nucleotide triples, that code for protein synthesis, are called codons. Introns are the portion of nucleotides that doesn't code for protein synthesis.

After the transcript process, the resulting sequence,the mature mRNA, is decoded by the Ribosome, in a process called translation. The goal of this process is to produce a specific amino acid chain, the polypeptide. The translation process start always with a start codon (most common start codon is AUG) and is not complete while the formed chain does not face a stop codon (UAA, UAG or UGA) (figure 2.3).

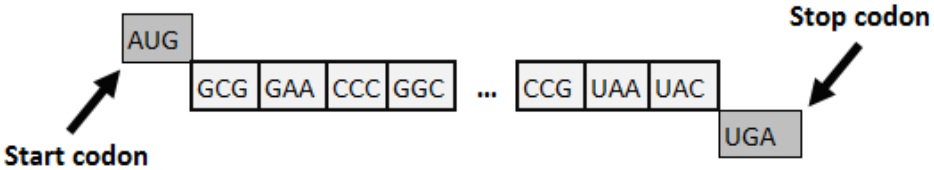


Figure 2.3: Example of the resulting amino acid chain provided by the translation process. As shown, this sequence always starts with a start codon and ends with a stop codon.

Note that a codon is nothing more than a sequence of three consecutive nucleotides from the mRNA sequence. Hence, the complete sequence must have always a multiple of three nucleotides length. After the translation process is complete, the resulting polypeptide chain will fold into an active protein.

2.2 Synonymous codons

In the previous section it was given a brief overview of the translation process. That process involves a set of twenty amino acids and each amino acid is encoded by a codon (sequence of three nucleotides). It was stated before that there are four types of nucleotides (adenine (A), guanine (G), cytosine (C), and thymine (T)) meaning that we can have up to sixty four different codons. Because there are 64 possible codons and only 20 amino acids (plus stop codons) some codons may encode for the same amino acid, being called synonymous codons. This characteristic is often referred to as the redundancy of the genetic code. Synonymous codons are easily found in the "Genetic Code Table", that displays the information about each amino acid and the associated list of codons who encode it. The table 2.1 represents the standard genetic code:

	U	C	A	G	
U	**UUU** - *Phe*	**UCU** - *Ser*	**UAU** - *Tyr*	**UGU** - *Cys*	U
	UUC - *Phe*	**UCC** - *Ser*	**UAC** - *Tyr*	**UGC** - *Cys*	C
	UUA - *Leu*	**UCA** - *Ser*	**UAA** - *Stop*	**UGA** - *Stop*	A
	UUG - *Leu*	**UCG** - *Ser*	**UAG** - *Stop*	**UGG** - *Trp*	G
C	**CUU** - *Leu*	**CCU** - *Pro*	**CAU** - *His*	**CGU** - *Arg*	U
	CUC - *Leu*	**CCC** - *Pro*	**CAC** - *His*	**CGC** - *Arg*	C
	CUA - *Leu*	**CCA** - *Pro*	**CAA** - *Gln*	**CGA** - *Arg*	A
	CUG - *Leu*	**CCG** - *Pro*	**CAG** - *Gln*	**CGG** - *Arg*	G
A	**AUU** - *Ile*	**ACU** - *Thr*	**AAU** - *Asn*	**AGU** - *Ser*	U
	AUC - *Ile*	**ACC** - *Thr*	**AAC** - *Asn*	**AGC** - *Ser*	C
	AUA - *Ile*	**ACA** - *Thr*	**AAA** - *Lys*	**AGA** - *Arg*	A
	AUG - *Start/Met*	**ACG** - *Thr*	**AAG** - *Lys*	**AGG** - *Arg*	G
G	**GUU** - *Val*	**GCU** - *Ala*	**GAU** - *Asp*	**GGU** - *Gly*	U
	GUC - *Val*	**GCC** - *Ala*	**GAC** - *Asp*	**GGC** - *Gly*	C
	GUA - *Val*	**GCA** - *Ala*	**GAA** - *Glu*	**GGA** - *Gly*	A
	GUG - *Val*	**GCG** - *Ala*	**GAG** - *Glu*	**GGG** - *Gly*	G

Table 2.1: Standard Genetic Code table.

One of the main objectives of optimizing genes for heterologous expression, is to achieve an improved nucleotide sequence. This improvement results from a process called *synonymous substitution*, where codons can be replaced by other equivalent codons (see table 2.1), without changing the resulting protein sequence. When a replacement occurs, the change is generally neutral. This means that these changes will not affect the protein that is produced. A simple example of the this process is shown in figure 2.4.

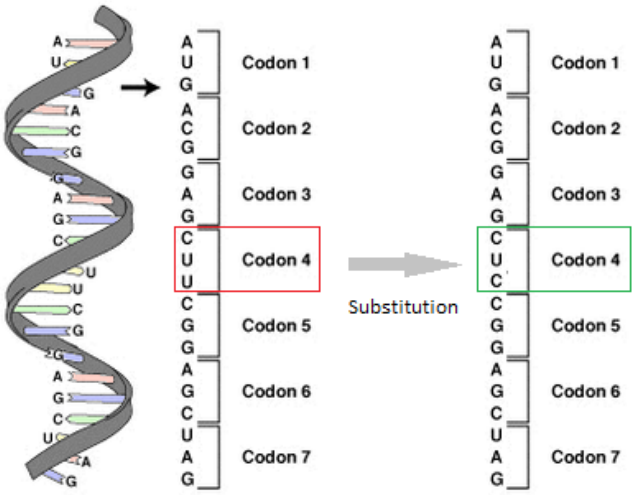


Figure 2.4: Substitution of a codon (CUU) by a synonymous one (CUC), both coding for the amino acid *leucine*, that does not change the resulting protein sequence.

2.3 Codon optimization

In section 2.1, a brief description regarding the translation process was presented. However, this process is a lot more complex. To fully understand the codon optimization techniques, it is necessary to know more about the translation process, especially the role of tRNA (transfer ribonucleic acid). The tRNA can be described as a linkage bridge between nucleotide sequence and the amino acid sequence of proteins. Hence, it can be described as a carrier that carry the correct amino acid to the mRNA on the ribosome during the translation process as shown in figure 2.5:

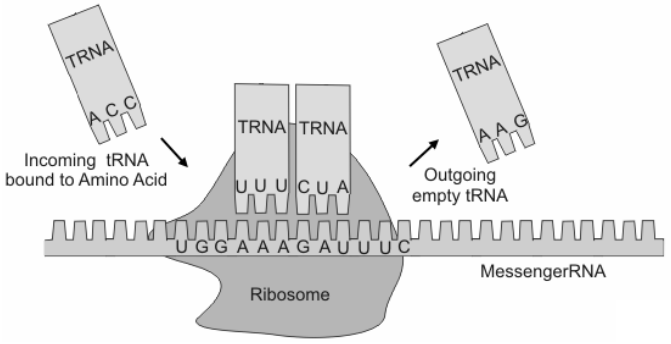


Figure 2.5: Gene translation process where the mRNA sequence is decoded by the ribosome to synthesize a protein.

This information is important for the following sections because codons take a major role in proteins improvement (gene redesign) as described in section 2.2. Next, several characteristics of the mRNA with relevance for gene optimization will be presented.

2.3.1 Rare codons

A low-usage codon is defined as a codon that is used rarely or infrequently in the genome. A rare codon is not only used rarely in a genome, since codon usage and tRNA abundance are usually well correlated, but is also decoded by a low-abundant tRNA [8]. This brings problems to the translation process. The translation rate for a rare codon is much slower than the one for a more abundant codon since the tRNA availability is lower for rare codons. One approach to tackle this effect, is to firstly identify the rare codons. Thus, it is assumed that a codon can be considered rare if it appears less then 5 times out of 1000 in a ORFeome of an organism. Note that this threshold to consider what is a rare codon can be higher or lower.

However, despite a codon being rare, it can be required for the gene expression and thus should be kept. One way to determine if this is the case is to do an orthologous comparison because, if the rare codon is required for a given position of a gene, that position should be filled by that rare codon, not only in the gene of that organism, but also in its orthologs. Hence, if the codon is considered rare and appears in the same position of the gene orthologs, or this positions are occupied by other rare codons, it should be kept, otherwise, it should be replaced by a more frequent codon.

2.3.2 Codon correlation effect

In the previous section, was stated that numerous tRNAs can compete with each other, at the acceptor site of ribosomes, until the correct tRNA is selected. This competition can make the process slow, reducing its efficiency. Thus, as well as in the "ramp effect", multiple tRNAs can encode more than one synonymous codon so reducing the number of needed tRNAs [5]. Hence, the speed of translation can be improved as well as the complexity behind all this process. This process can be applied to the entire gene, trying to avoid as much as possible the switch of tRNAs, by using the same synonymous codon along the gene sequence. The translation speed can be greatly improved since the tRNAs do not need to disperse in order to get charged again without leaving the ribosome vicinity (figure 2.6).

This effect can be more important if a high level of expression is required. Nevertheless, one should be aware that synonymous codons substitutions that change codon usage frequencies from infrequent to frequent, in regions with slow mRNAs translations, can deleteriously affect the protein quality, mostly because this can exhaust the tRNA pool of the cell, causing an overall imbalance between codons and their cognate tRNAs [6]. One possible solution to override this problem would be knowing how many tRNAs are available. As a consequence, codons decoded by abundant tRNAs need to be more frequent than their synonymous [7].

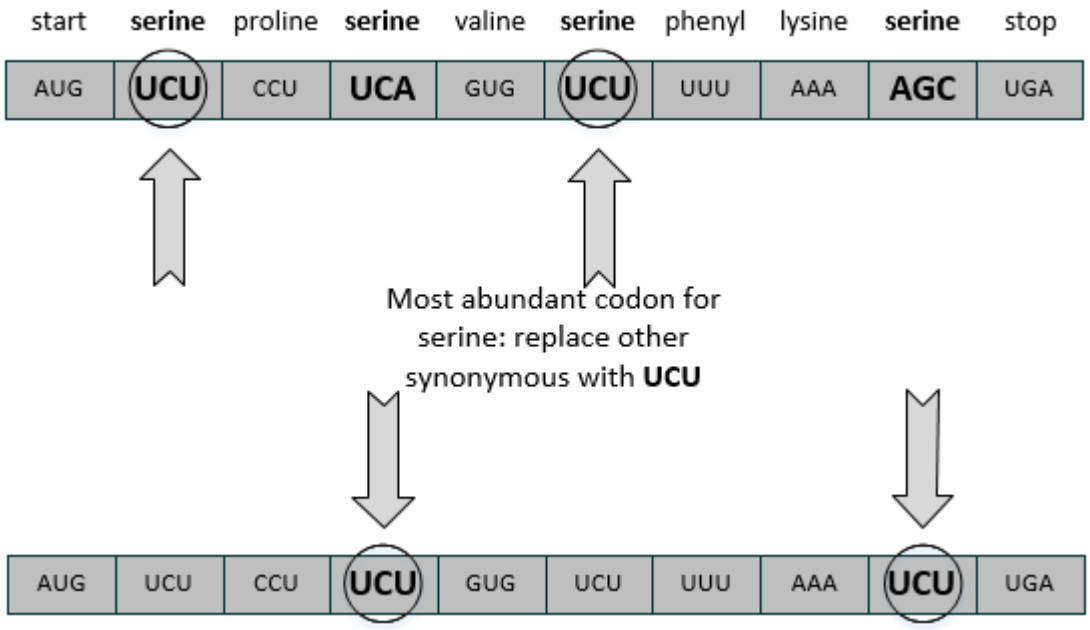


Figure 2.6: Codon Correlation Effect process - The gene is evaluated choosing the codon who appears often in the gene for each amino acid. After that all other codons for the same amino acid are replaced with the codon who appears more. This process will force the use of the same tRNA increasing the translation process

2.3.3 Ramp effect

The translation process is not in any way linear. mRNA translation evolves multiple stages and many mechanisms that are complex and hard to detail. Furthermore they go out of the outline of this thesis. However, the translation process is critical for gene expression and it must be as much as possible efficient. The abundance of charged tRNAs that correspond to the different codons that encode a protein, was suggested to determine the speed and accuracy of translation [3]. This means that codons have a unique role in translation process and the abundance, or stringency, of tRNAs determines the high and low efficiency respectively. In other words, we can specify that abundant codons are decoded with high efficiency, because the are better adapted to the tRNA pool. With this information, we can define two different types of translation during the translation process. The first, that will be named slow region, complies the codons where we have different tRNAs to encode the same amino acid. In this case synonyms codons are avoided resulting on a slow translation since every codon needs to wait for a new tRNA. The second type, that will be named fast region, is where we can encode the same amino acid with the same tRNA, if it is a synonym. Thus, this process relates the abundance of tRNA and the amino acid translation speed and accuracy [4].

Codons can be translated at different speeds as stated before and this is because the frequency of codons is directly correlated with tRNA abundance. Thus, if there are several rare codons at the beginning of a gene sequence, they will be translated at slow speed since rare codons are usually decoded by rare tRNAs [3]. Therefore, the redesign efficiency of the translation process can be optimized using this knowledge. The main idea is to use rare synonymous codons at the beginning of the gene, so each codon is translated by less abundant

tRNAs which takes more time. Furthermore, the area of effect of this "ramp effect" can be defined, for instance, for the first 30 codons. After this region, the codons should be translated normally and the ribosome congestion should ever be avoided, or partially avoided.

2.4 EuGene - Gene Redesign Software

In the previous sections, several processes that can improve a protein expression were discussed. Those processes are connected to the translation process, which is quite sensitive, and they all try to achieve a common goal which is to synthesize a protein faster, without changing its quality and functionality. As a result, several informatics solutions have been developed that have a huge role aiding this processes. These solutions are widely used in different fields. For instance, improving the protein production can be very important in the development of new vaccines as well as in their manufacture [9].

In this section, a specific gene optimization system is explained. Next, will be presented the main characteristics of the system as well as how it can be explored even further, using new algorithms as well as a new application for those gene redesign methods. Furthermore, a deep analysis is made regarding the performance of the overall system and where it can be optimized.

2.4.1 Brief description

As described above, EuGene explores expert algorithms to redesign genes for heterologous expression. Thus, besides redesign methods, EuGene display several informations about a specific selected gene. Furthermore, to identify a gene, EuGene can use FASTA[1] and GenBank[2] formats to extract any database identifiers. Those are then used to access NBCI and obtain gene and genome names and also the resulting protein sequence [10]. After the gene selection, EuGene displays in an intuitive way the gene sequence as well as the amino-acid sequence and relevant information about that gene (figure 2.7.

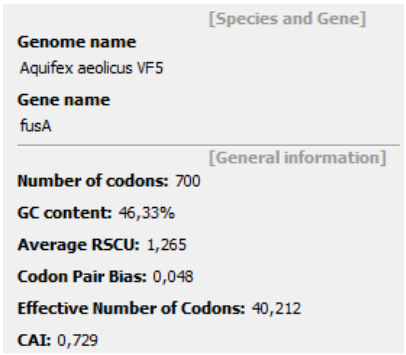


Figure 2.7: EuGene quick information about a single gene - CAI, RSCU, codon pair bias and some other useful information

[1]FASTA format is a text-based format for nucleotide or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes.

[2]GenBank format (GenBank Flat File Format) consists of an annotation section and a sequence section. The annotation section has information about the gene, like organism, definition etc and the sequence sequence has the gene nucleotide sequence.

In a seamless automatic form, EuGene calculates the "CAI" (Codon Adaptation Index), "RSCU" (Relative Synonymous Codon Usage), "CPB" (Codon Pair Bias, for codon context), number of codons and also the "GC Content" (quantity of Guanine and Cytosine pairs) of the opened gene. This pre-calculated information is an important attribute for the redesign methods, so each method do not need to re-calculate this information, resulting also in a performance tweak.

Hence, EuGene has a set of functionalities that cooperate between them in order to achieve a good performance and result, and also a display of the set of relevant actions that are rather transparent to the user. Figure 2.8 presents an overview of the system.

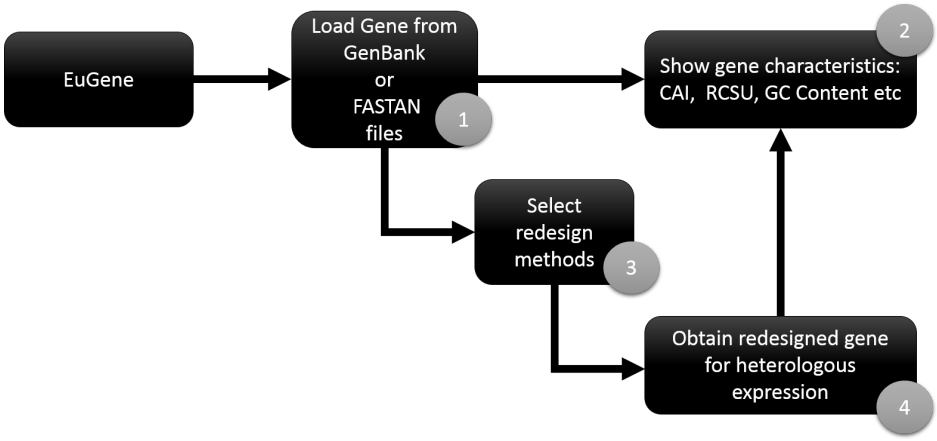


Figure 2.8: EuGene overall operation mode schema: 1 - Possibility to load FASTA and GenBank file formats to retrieve information about a genome and its genes; 2 - Displays information about a specific selected gene (CAI, RSCU...); 3 - Selection of several optimization methods to improve the protein expression (subsection 2.4.2); 4 - Observe results from the optimization methods, including the new codon sequence as well as information about the score achieved by each method.

In conclusion, this section presents some features of EuGene in a summarized way, opening ways to the next big feature, the redesigning methods, that will be detailed in subsection 2.4.2.

2.4.2 Redesign methods

The main feature presented by EuGene is the capability to optimize codon sequences using a combination of several different approaches. Those comprehend a set of algorithms that allows the customization of a gene following specific redesign algorithms. Table 2.2 shows which codon optimization methods are currently available to use in EuGene. Note that this table does not try to explain in detail how every method implemented in EuGene works, but it gives a brief overview about their propose in the context of gene redesign.

Optimization redesign method	Brief description
Codon Usage	Allows to maximize or minimize the codon usage. This is done by counting the number of times each of the 64 codons appears;
Repeats Removal	Replaces zones with repeated nucleotides with synonymous codons breaking the repeated sequence chain;
GC Content	Calculates the percentage of GC nucleotides. For instance to maximize this effect every codon is replaced by a synonymous with higher GC amount;
Codon Context	Calculates the number of times a pair of consecutive nucleotides appears. Thus, optimizing by codon context is finding the synonymous codons that maximize the frequencies of those consecutive pairs;
Site Removal	Remove specific nucleotide sequences, for instance Kozak sequences (GCCACCAUGG), by replacing the codons within it by synonymous codons;
Hidden Stop Codons	Replaces out-of-frame stop codons, for instance, CC**UAA**C, by replacing two codons by synonymous in order to eliminate out-of-frame stop codons;
Unmodified tRNAs	Avoid a specific set of codons for Eukaryote or Bacteria. These codons are decoded by less efficient tRNAs;
RNA Secondary Structure	RNA secondary structure prediction in order to eliminate double strand occurrences;

Table 2.2: Gene optimization methods available in EuGene. A set of different approaches are presented as well as some brief information about each method.

To use this redesign methods, in order to achieve a good optimized sequence, EuGene uses two distinct optimization techniques: Genetic Algorithm[3] and Simulated Annealing[4]. These two methods are used in distinct situation, depending if the user wants quicker results or deeper results. Simulated Annealing is a faster algorithm that tries to find an optimal solution quicker but with less precision. On the other hand, Genetic Algorithm search deeper, making it slower, not only for one optimal solution but for a list of possible best equivalent solutions, allowing selection of the solution that offers the best trade-off between the selected redesign methods [10].

[3]The Genetic Algorithm is an adaptive strategy and an global optimization technique. It is an Evolutionary Algorithm and belongs to the broader field of Evolutionary Computation [11]

[4]Simulated Annealing is a global optimization algorithm that belongs to the field of Stochastic Optimization and Metaheuristics [11]

2.5 Summary

This chapter described a brief introduction to the biological process behind the protein synthesis. Moreover, a definition of DNA was presented as well as its contents, genes, that are used in the protein synthesis. This process is described in all its steps, since how a gene is copied from DNA until a chain of amino acids is formed and after becoming a protein.

Furthermore, it is explained how genes can be redesigned by substituting codons by equivalent ones that code for the same amino acid. This flexibility allows to explore several redesign methods with the purpose of increase the proteins qualities, also some of this methods were detailed. Finally, a software (EuGene) that is capable of join several gene redesign methods in a unique tool was presented. This software acts as the root for this thesis since its foundation rise from EuGene features.

Chapter 3

Requirements and Architecture

EuGene is a powerful tool that implements several redesign algorithms, as stated in chapter 2, and some of its features can be reused for another purpose. Therefore, this chapter presents how those features can be reused and also further detailed information about the requirements and the architecture that led to this project implementation.

3.1 User requirements

With a tool such as EuGene, that accommodate several tools in one single application, researchers notice that the evolution of genes could maybe be explained, if they could predict what kind of mutations led to that evolution [12]. The redesign methods available in EuGene, can hence be combined in such way that allow explaining how a gene evolved to a synonymous one.

Overall, using such redesign methods, become clear that a new tool could be developed to evaluate how a gene evolved to its current state. This was the main goal of this project, i.e, to combine several redesign methods, and their parameters, and try to achieve as much as possible a new gene with high similarity to its current state. This computational process takes time to complete because several redesign methods needs to be combined, and evaluated, until an optimum solution is found. Moreover, computing time increase as the gene length increases, and therefore, the performance provided by the system should be optimized to mitigate high execution times. As extra requirements, some new features will be implemented to enhance EuGene functionality.

The following sub-sections describe the most important user requirement, as well as the architecture behind the application.

3.1.1 General objectives

Gathering a complete set of requirements is the most important step at the beginning of any software project. Thus, the most important requirement relies on the ability to generate random synonymous genes from a specific gene and then apply several combined gene redesign methods to each. After all those methods have been applied, it is necessary to evaluate the score of the resulting sequence, for each synonymous, in comparison with the original one. These scores can be obtained by using known techniques that can tell how different the sequences are, for instance providing a similarity percentage, like *Hamming Distance*. The

whole process is detailed in chapter 5. Finally, the optimization of the execution time is a second major requirement.

3.1.2 User interface

When humans interact with a machine, an important role relies on the user interface. A good interface should provide a "user friendly" experience allowing users to completely understand how they can use the application.

With this in mind, a first interface draft was needed, which layout should accommodate most of the required features such as plugins selection, genes and results display. Based on EuGene application, a new interface mock-up was built resulting as shown in figure 3.1:

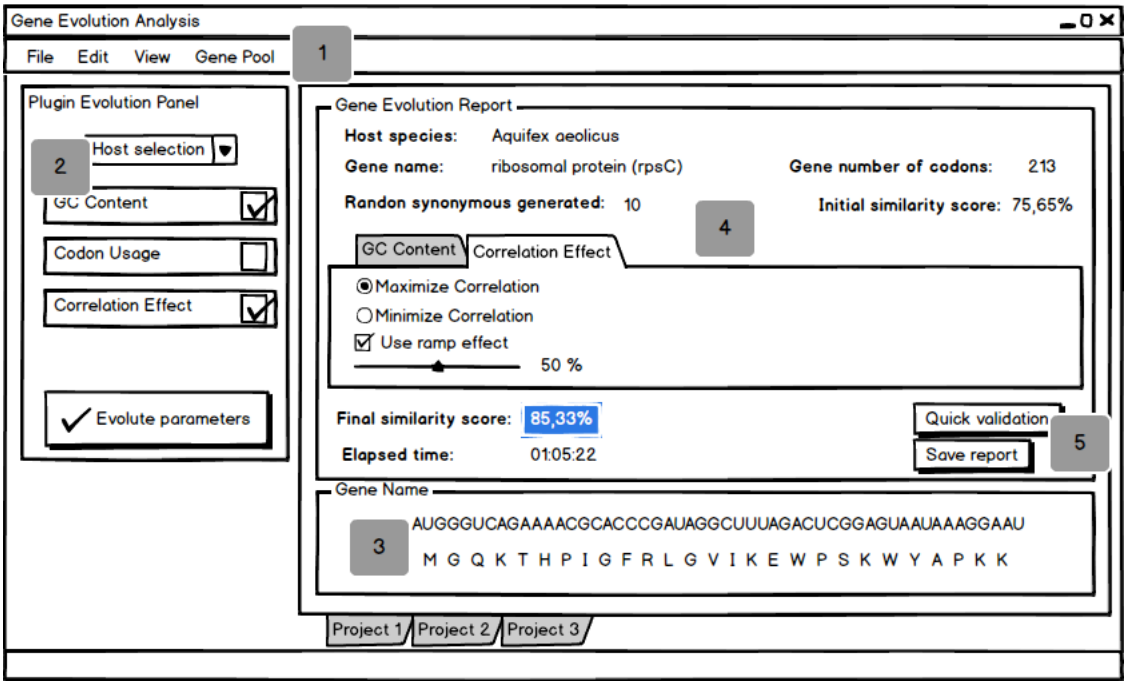


Figure 3.1: Basic user interface mock-up to accommodate the main features

The main reason behind this similar interface is that the resources used by this project application are the same used by EuGene, i.e. gene redesign methods that can either be used in this project application. Thus, in this illustration, numbers show the main parts of the application layout and the ones which need further detail.

The first part (number one), represents the menu bar. This bar should have four distinct menus: *File, Edit, View, Gene Pool*. The *File* menu is intended to provide functionalities such as create a new project or load past projects. Loading a saved gene is important to check what kind of mutations are responsible for gene evolution. On the other hand, the *Edit* menu provides a set of settings to improve the algorithm accuracy. Those include the number of random synonymous genes to be generated as well as the maximum number of iterations the algorithm can take to converge. The *View* menu is an extra option just to enhance user interaction experience, by showing or hiding panels. Finally, the *Gene Pool Menu* is intended to load genes from files or even insert a gene manually.

The second zone (number two), is a panel that should include all optimization methods

as well as the option to select the desire ones. Here it should also be possible to see what kind of parameters each method use. This zone also include the *Run* button that allows that application to start evolve the optimization methods to achieve a final result.

In the project area we can see two distinct zones. One regarding the gene representation (3), where the user can see the nucleotide sequence as well as the amino acid sequence of the loaded gene. The second zone, represented by number 4, shows the final results achieved. Here the user should be able to see information such as how many random genes where used as well as the best parameters combination used to achieve the final result. Also the final similarity score obtained using those methods and parameters should be shown, as presented in panel marked with number 5. This zone also has the option to make a quick validation. This enables the user to test the achieved parameters with new random genes or even insert synonymous gene sequence as shown in figure 3.2:

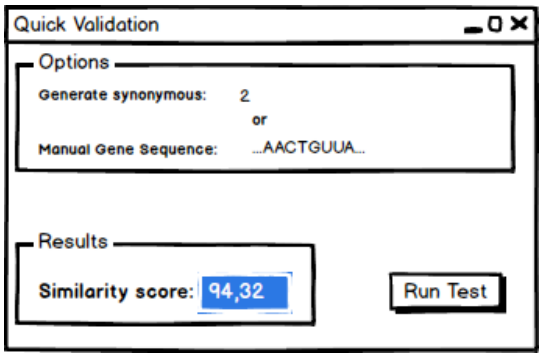


Figure 3.2: Quick evolution parameters analysis mock-up

This mock-ups represents a quick draft of how the basic application functionalities should be represented. The final results may change due to new specifications or even performance issues.

3.2 Functional requirements

From the software engineering point of view, functional requirements are defined by the objective/function of the software system, and hence, they should describe what data the system should complies and how it is used. Moreover, functional requirements are supported by non-functional requirements (section 3.3) which impose some constrains like portability. Therefore, there were identified some crucial requirements that the system should have.

Firstly it is necessary to be able to manipulate gene sequences, namely opening and parsing a genome file and display the gene sequence (codon and protein structures). This requirement is inherited from EuGene since it has a well defined module called *Gene Pool* which takes care of the genome load and manipulation.

Other requirement is the capability of use the developed gene redesign methods that EuGene currently supports and also use new redesign methods, *Keep Rare Codons, Codon Correlation Effect and Ramp Effect.*

Finally, the system should be able to explore all redesign methods applying them to random synonymous genes generated from a wild type gene (original gene), in order to achieve the best set of parameters that lead the synonymous to evolve, as much as possible, in a new

sequence similar to the wild type. This can be defined as one of the main requirement of this thesis. Moreover, having such tool that can produce this results in lesser time is crucial and therefore the redesign methods performance should be tuned as well as the global optimization process, detailed in chapter 4.

This main requirements, as well as the interface needs are presented in table 3.1.

Requirement	Description
Gene load	
Load FASTA file	Open genome file, FASTA format, and select gene
Load GenBank file	Open genome file, GenBank format, and select gene
Add Gene Manually	Add custom gene manually for a specific genome
Gene Sequence Optimization	
Plugins provided by EuGene	Use EuGene gene redesign methods to optimize genes
Optimization by Rare Codons	Section 2.3.1
Optimization by Correlation Effect	Section 2.3.2
Optimization by Ramp Effect	Section 2.3.3
Combinational System	
Generate synonymous pool	Create random synonymous genes to a native selected gene
Find best redesign parameters	Explore optimization methods selected by the user to find the best parameters combination using Simulated Annealing algorithm to find a global maximum (best parameters)
Evaluate results	Create a report with the best parameters and the resulting score
Cross validation	Make a new validation of the resulting parameters with new random synonymous genes
Interface	
Display optimization methods	Allow user to see which gene redesign methods are available
Save gene evolution report	Save the resulting data
Load gene evolution report	Load a previous saved gene evolution report
Optimization Methods	
Optimize gene redesign methods	Explore new techniques, and new algorithms, to reduce the gene redesign methods execution time
Optimize global optimization system	Analysis of the optimization system, finding how it can be improved to offer the same results in lesser time

Table 3.1: List of main requirements that the application should support

With all these requirements, it is noticeable that exists a crucial requirement: combination of several redesign methods. The next sub-section provide additional information about it.

3.2.1 Combinatorial system using gene redesign methods

The main feature of the system relies on the combinatorial system. This system can be seen as a box that receives a gene and tries to give hints of how this gene evolved. That process encompasses generating a pool of synonymous and then applies to each synonym

several redesign methods. Each of these methods should try to achieve the highest score as possible, i.e, increase the protein quality as much as possible. Though, each method have a set of parameters, for instance, the codon correlation effect have the possibility to increase the correlation between codons or even decrease this correlation, as explained in section 2.3.2. These parameters should be tested in different combinations for all redesign methods in order to achieve a final result - maximum similarity of each synonymous and the original gene. This process can help explaining what factors led a gene to evolved to its current state.

Note that due to the different possibilities of combining the redesign methods parameters, this process is exhaustive, and therefore it takes long time to complete. On the other hand, once a gene is evaluated, it no longer requires further evaluation since the set of parameters achieved by each redesign methods is final. A simple draft of how this system works can be seen in the figure 3.3:

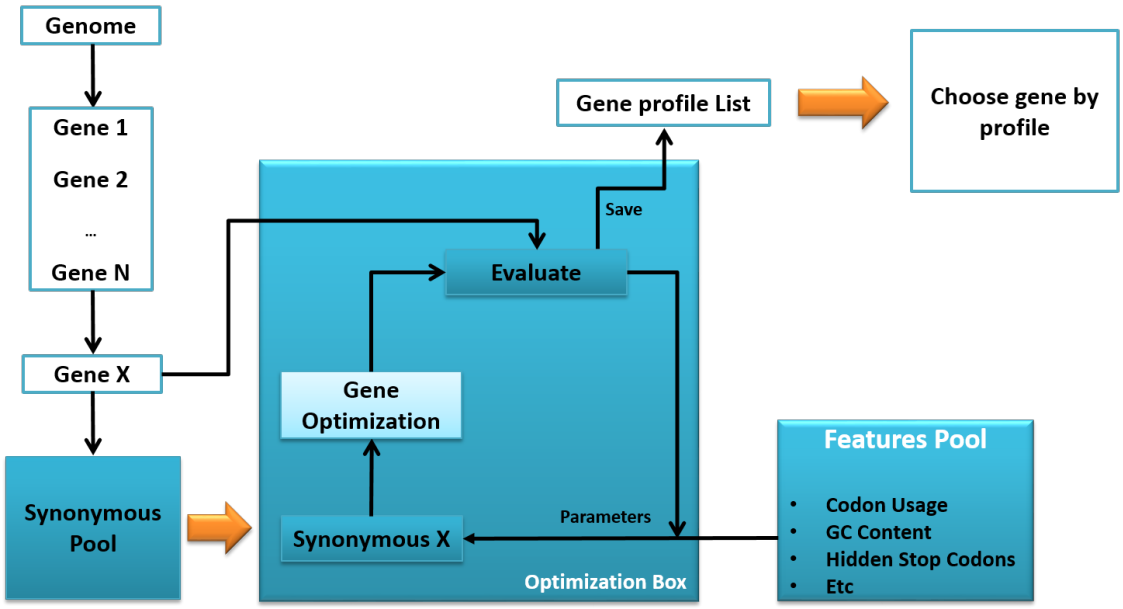


Figure 3.3: Optimization system draft displaying the system flow. First a genome is chosen and from it is selected a gene. After, a pool filled with synonymous genes is generated, and for each synonymous a set of redesign methods are applied, iteratively, until an optimal result is found

3.3 Non-functional requirements

3.3.1 Portability

Nowadays, one of the biggest bottleneck in software applications is the portability. This means that independently of the operating system, processor or machine, the application should run in all environments. Hence, using Java programming language solves this issue given its portability for different operating systems. Thus, this project software should run without any problem in any environment with Java, for instance, Linux, Windows or Mac OS.

3.4 Summary

In this chapter, the main requirements for this thesis were presented. After reading this chapter it should be possible to understand the need to generate random genes from a native one, and process them in order to achieve a similar gene to the native. This process is done by applying several redesign methods with random parameters and then applied a metric like *Hamming Distance* to evaluate the similarity. Moreover, an interface scratch is presented that shows how all the requirements should fit in the application. This application, inherit from EuGene most of its visual contents as well as new features to support the functional requirements. These requirements were presented in a compact table allowing a quick understanding of what work needs to be done to accomplish this thesis goal. Furthermore, the need to have an application that can run in distinct operating systems was presented in order to ensure portability.

Chapter 4

Optimization

Chapter 3 presented main requirements to achieve this project goal. Thus, being an exploratory work, the performance of the existent redesign methods, as well as the optimization process, must the tackled. Those methods, implemented in EuGene, makes use of a simple approach where the execution time does not matter a lot. For this project, where those methods are going to be used repeatedly, a more deep analysis was performed in order to see how their execution time can be decreased, or at least for most of them. Moreover, this chapter also explores the *Simulated Annealing* process, presented in EuGene, and a possible solution to improve its performance.

4.1 Finding the bottleneck

In any computer program, there are some code zones that are executed more frequently than others. Despite being more frequently executed, does not entirely means that they are slowing down an entire process and thus, finding the critical spot, *bottleneck*, is a meticulous task. Therefore, the optimization process presented in EuGene, was carefully analyzed and divided into three distinct blocks - ***Simulated Annealing entire process***, ***codon neighbour generation*** and ***plugins exectution***. The goal of this division is to find where the optimization process can be improved. Thus flowchart (4.1) shows how this division was performed. Note that these are the most important blocks in the SA algorithm. For the previous blocks presented, an average execution time was measured resulting in the following table 4.1:

Number plugins	Total Time	Plugins execution	Neighbour generation
3	85588	84123	147
5	67721	66183	172
7	120415	118672	179

Table 4.1: Block times using random *plugins* - This results were achieved using distinct *plugins* in each case and with a gene with 1000 codons length. Other tests were made using genes with distinct lengths and the results were similar. Also, times are presented are in **milliseconds**.

As observed, it is clearly that the *bottleneck* remains in the *plugins* execution environment. They use almost all the time of the Simulated Annealing algorithm and, therefore, is the perfect spot to perform a deeper analysis (section 4.3). These results showed that the

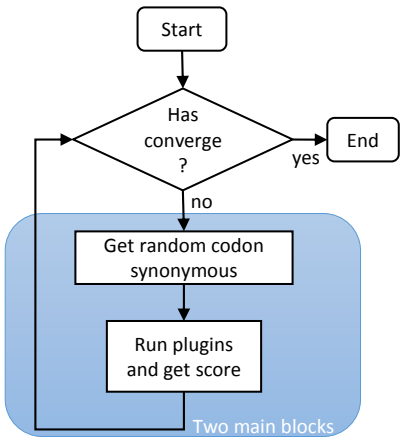


Figure 4.1: Simplified version of the SA flowchart showing the most important blocks

average consumption time of the *plugins* execution took around **98%** off the entire Simulated Annealing process.

4.2 Simulated Annealing

The redesign methods execution takes almost all the computational time of the Simulated Annealing algorithm. Hence, before checking how this algorithm can be improved it is important to understand how it works and what it tries to achieve.

Over the last decades, many optimization heuristics have been developed. Those have the objective to find the optimal solution for a given problem. Therefore, improvement heuristics start with an arbitrary configuration for a given problem, and tries to improve its solution iteratively by changing small pieces of the original configuration and evaluating the new one.

A simple and well known heuristic is the Greedy Algorithm. In short, it move pieces of the initial configuration, randomly, and only accept the changes made by the move if the new configuration is better than the previous one. On the other hand, if the configuration is worst, it is discarded and one stays at the "previous best" configuration.

However, using such approach can lead to stuck situations where the solution found is a local maxima but not a global one. When this happens, the system can not improve the solution. To overcome this problem, it is necessary to work with more acceptance functions and not only choosing pieces randomly. Note that more acceptance functions do not free the system to get stuck in local maximas, but reduce the probability of that to happen[13]. One used optimization algorithm that follows this approach is the Simulated Annealing.

Every redesign method that is present in EuGene returns a *score* for each sequence, according to the redesign algorithm objective. This allows Simulated Annealing to perform an aggregation of all the redesign methods scores and therefore perform an evaluation by checking if the global score achieved is better than the previous one. Note that in the gene context, each synonymous sequence of the original gene sequence represents a possible solution.

Hence, Simulated Annealing works according to the choose of several candidate solutions, even if its worst than the current, and evaluate its final score. As the iterations goes by, Simulated Annealing choose only the best solutions (it does not guarantee the optimum solution). This process can be referred as *slower cooling* since initially Simulated Annealing

will find good solutions, and, with time, it will become harder to find better solutions than the previous obtained[14].

The original implementation of this algorithm (in EuGene), uses *Strings* to manipulate and create new gene sequences. Most of the effort of the algorithm, besides the effort used by the gene redesign methods, is used in obtaining a new random neighbour (codon) and apply it to the current best sequence. In order to improve this solution, every gene sequence was converted to an *Array* of *Integers*, where each position uses an *ID* that identifies a specific codon. This allows to manipulate the gene sequence by just using array references. Using this approach, not only makes the Simulated Annealing algorithm to generate new sequences faster, but also aid the gene redesign methods in the sequence manipulation, since most of those methods operates at codon level.

A small test was made to prove how using an array of integers can improve the overall performance. Using *Java* programing language, it was created a *StringBuilder* with 4725 characters and an array of integers with 4725 positions initialized with random integers. Then, it were made 100000 iterations over those variables according to the following:

- **sequence1.replace(randomIndex, randomIndex, "A");** for the StringBuilder
- **sequence2[randomIndex] = 1;** for the integer array

The measured time was at milliseconds scale with the results of 453 and 9 respectively. Despite being a small test it shows how faster it is to attribute a value to an array than replacing a specific *Character* in a *String* sequence.

This approach can certainly improve the overall Simulated Annealing algorithm as well as similar operations in the other gene redesign methods.

4.3 Plugins Analysis

Manipulating several data structures can be a hard task since each one have a unique behavior. However, due to the plugin structure adapted in EuGene, each redesign method can be tackled independently. From this problem point of view, each redesign method uses a unique algorithm and therefore their behavior is distinct.

Achieving an optimal result takes time and the algorithm complexity plays a major roll that needs to be tackled. Hence, as first step, each *plugin* was analyzed individually. The purpose of this analysis is to measure the average time each *plugin* takes to achieve the best result and therefore know how the system overall performance is affected. For this test scenario a couple of tests were made, using two distinct genes lengths. Those lengths may define the convergence time of each plugin, however, other factors like the gene codon sequence, may fit better in one plugin algorithm than other, and therefore the algorithm does not need to put the same effort as it should in a more "scrambled" sequence. In other words, if for instance a hypothetical redesign method had the objective of removing the base **A**, and the codon sequence does not have any **A**, then a single iteration would be necessary to perform such task and the optimal solution is easily reached. However, if a sequence has several **A** bases, it would be necessary to remove each appearance, which would add more complexity to the algorithm by removing codons and replacing them by synonymous ones that does not have **A** bases. Hence, the gene length may not be the only factor affecting the algorithm performance.

In these initial tests, and to measure the time each *plugin* takes, only the gene length was taken in consideration. This is enough to find which *plugins* are performing bad in a general case scenario. Hence, four distinct genes were tackled from two different genomes - **Escherichia Coli** and **Aquifex Aeolicus**. From each genome, two genes were randomly chosen, one with a codon length of **200** and another with **600**. Also, each *plugin* was run five times for each gene and an average execution time was taken. The purpose of taking a mean time is to avoid an inefficiently convergence of the Simulated Annealing algorithm that can sometimes not achieve the optimal solution. The following execution times, in milliseconds, were observed:

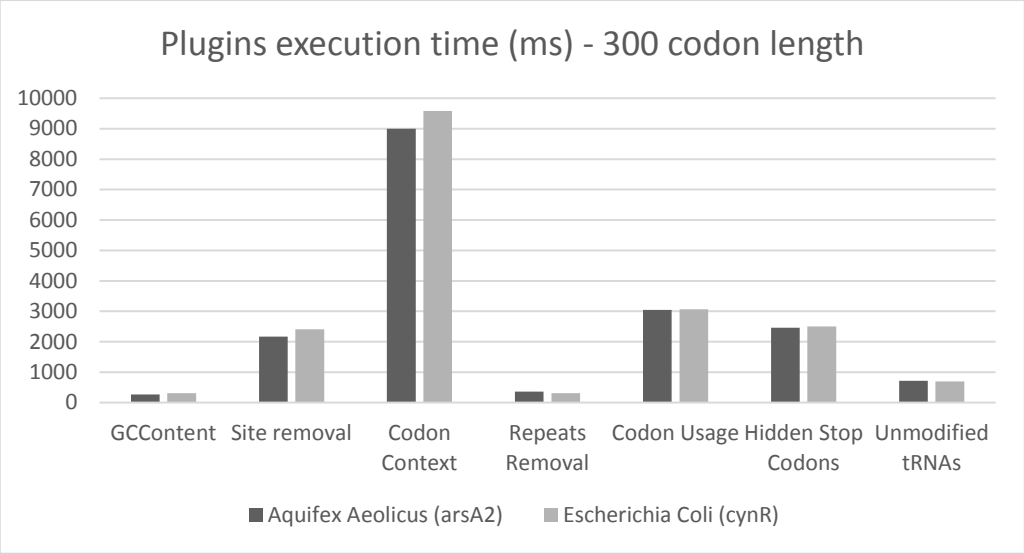


Figure 4.2: Plugins execution time of two distinct genes with 300 codons length

Chart 4.2, shows that at least four *plugins* have a major impact in the global performance. For a small gene, with 300 codons, the redesign method ***Codon Context*** can take up to approximately **9,5 seconds** which is pretty high. For instance, if the combinatorial system presented in chapter 3 took up to 500 iterations to find the optimal parameters, meaning that it would need to evaluate 500 different parameters for this plugin, then the required time for completion would be around: $9,5 * 500 = 4750s$, which, for a single plugin, would result in a global execution time of approximately **80 minutes**. Other plugins, like *Site Removal, Codon Usage and Hidden Stop Codons* also take a considerable amount of time to complete as we can see in the previous chart. Chart 4.3, presents similar results for a bigger gene. It is noticeable that the same *plugins* have a higher execution time, proving that they are a *bottleneck*.

From the available *plugins* provided by EuGene, one is missing in this tests - ***RNA Secondary Structure*** prediction redesign method. This one has a much higher execution time being that the reason why is not presented along with the other *plugins*. The chart would not be easily read since it would present a huge discrepancy between the *RNA Secondary Structures* and the other *plugins*. However, table 4.2 shows a direct comparison between this plugin and the worst one presented before, *Codon Context*.

As presented, the *RNA Secondary Structure* takes, at minimum, **10 times more** to

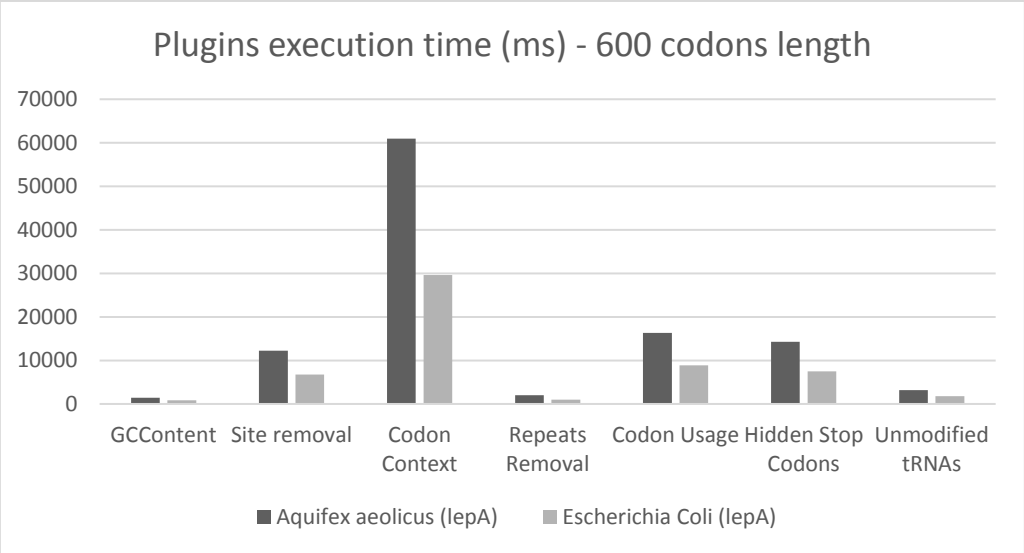


Figure 4.3: Plugins execution time of two distinct genes with 600 codons length

	RNA Sec. Struct.	Codon Context	Time Difference
	Genes with 300 codon length		
Aquifex Aeolicus (arsA2)	98043 ms	9002 ms	∼11x more time
Escherichia Coli (cynR)	101716 ms	9578 ms	∼10x more time
	Genes with 600 codon length		
Aquifex Aeolicus (lepA)	1061091 ms	60970 ms	∼17x more time
Escherichia Coli (lepA)	607737 ms	29638 ms	∼21x more time

Table 4.2: Difference between RNA Secondary Structure and Codon Condext - The *Time Difference* column shows how many times the Codon Context plugin could run until the RNA Secondary Structure plugin achieve the best result.

execute than *Codon Context* being this the main reason why it is not present in the charts. It would not be possible to see clearly the average time that all plugins takes to execute and, therefore, the need to make improvements at this level. Despite being the *plugin* that takes more time to execute, it does not make it the unique plugin that needs a performance boost. Every other *plugin* may be, and must, be improved.

Looking at *RNA Secondary Structure* execution time, it is noticeable how long it takes to execute. Note that this results were taken for a single *plugin* at a time, and if for instance, we run four plugins at once, they would need even more iterations to converge. The conjugation of several plugins can force some plugins to scramble the results achieved by other plugin, for instance, if one plugin was intent to replace all A bases by C and another plugin was intent to replace all C bases by A. This results in an even greater time needed to achieve the best result.

The execution time for a single iteration is long, and for multiple iterations, in a combinatorial environment, it could take weeks to achieve an optimal result. Since every *plugin* will be tested several times, with distinct parameters, every second that can be improved in the algorithms execution is important and therefore it should be the next step taken before advancing to the development final application (chapter 5).

4.4 Plugins optimizations

Section 4.3 showed how important it is to make optimizations. Hence, several distinct strategies can be applied to each *plugin* according to its behavior. Previously, it was shown that EuGene is written using *Java* programming language and, being this language an *Object Oriented* one, objects cost time and CPU on their creation. One approach to improve *plugins* execution time could be holding some objects in memory since managing them in memory is much faster than processing them over and over again for getting the same result. This can free CPU to do another tasks while objects are being kept in memory. Moreover, avoiding unnecessary temporary objects, which takes time to create, can also affect an application performance. Note that every second that can be spare is important to achieve this thesis goals. On the other hand, some *plugins* may also be implemented using new algorithms that can be more efficient than the previous ones.

Using memory, there are some good programing practices that should be used. Memory leaks are possible in Java, like in any other programming language. For instance, it is possible to have memory leaks by holding on to objects without releasing their references. This usage stops the *Java Garbage Collector* from reclaiming those objects and therefore increasing the usage of memory being used [15]. This could lead to an inefficient usage of objects and therefore to an excessive memory consumption. Hence, next is presented a few guidelines that lead to a good programing and also to produce more efficient code:

- Avoid objects replication. This is important in routines where objects are used frequently. The over creation of the same object is unnecessary as well as inefficiently and also adds an overhead that can be avoided;

- Use static variables when multiple classes need access to the same object. It is preferable to use a static variable than have each class instance holding a separate reference for the same object. This also creates less variables since each class will use the same object without the need of creating one for each class;

- Avoid *String* exhaustive manipulation. On solution could be using integer IDs to identify strings. Integers are much less complex to manipulate and to compare;

- In a Thread environment, some methods must be synchronized. This synchronization should be applied to the method block that needs to prevent concurrent access and not the entire method;

- Using Threads can vastly improve system performance since it allows parallel computation;

These are just some examples of good practices that can lead to a more efficient and optimized code. More examples could be named but this list would be too extensive. However, one more example of a good practice should be referenced - *data structures* usage. Choosing the correct data structure to hold data can sometimes improve the system performance greatly, for instance, choosing between a *Vector* or an *ArrayList*. Despite being considered deprecated the *Vector* class may be suitable in some situations where concurrent modification matters. *Vector* has its methods synchronized granting concurrent security (mutual exclusion) while *ArrayList* does not. On the other hand, since ArrayList does not have synchronized methods, offers better performance and is more suitable for most applications. Note that it is possible to force an *ArrayList* to be synchronized using Collections - *Collections.synchronizedList(List)*. This small details can make difference in the overall system performance.

Likewise *Vector* and *ArrayList*, one often common problem, relies on the choose of the appropriate *Map* structure. *HashMap* does not have any method synchronized and hence does not offer object thread safe while *HashTable* offers synchronization in all methods. Also, being a direct consequence, *HashMap* presents higher performance while *HashTable* is slower.

In short, choosing the right data structure for a given application can vastly improve the system overall performance and therefore is a task that needs to be carefully thought depending of the system needs. With some of this ideas present, some modifications can be performed to the *plugins* algorithms but, before analyze each *plugin* individually a common modification was done.

These *plugins* were developed for EuGene and, curiously, EuGene avoids the usage of *ArrayLists*. Being a multi-thread application, specially using Swing[1], that is not thread-safe, the usage of *Vector* class was a good choice that could prevent most of the deadlocks that might appear. Note that Swing use a specific thread to manipulate his components **EDT** - *Event Dispatch Thread*. Hence, for data manipulation, where there is a need to insert and remove values, the *Vector* class offers thread safety, although, it may not be necessary if the concurrent access is well controlled, or even if it does not exists. Most of the operations used by the *plugins*, that use a *Vector* data structure, are *gets* (operation of retrieve a value from the vector). This kind of operation does not modify the vector data and therefore being in a *Vector* data structure provides an unnecessary overhead due to the synchronization of the class. Hence, *plugins Vectors* data structures were replaced by *ArrayList* to avoid this overhead. This substitution despite offers a performance improvement, also provides a better programing methodology since *Vector* class is already deprecated.

Besides this change at the *plugin* level, it was also made in all EuGene application classes. The usage of only *Vector* classes could bring down system overall performance and not only

[1]Swing is the primary Graphical User Interface provided by Java. Since Swing components are fully implemented in Java they are plataform-independent and therefore a good choice for use in distinct environments

at the *plugins* runtime. This change was carefully done since some vectors really needed to be synchronized - it was used the *Collections.synchronizedList(List)* to overcome this synchronization problems. Despite being out of this thesis outline, it was a simple task that enhance EuGene overall performance.

Finally, an individual analysis was performed to each *plugin* resulting in some changes that enhance their performance. Before those changes are presented, it is relevant to known how *plugins* are implemented generally. In order to develop a gene redesign method that can fit in EuGene, from the software developing point of view, it needs to respect a contract - *interface*. This interface forces the implementation of several methods that need to exists in the new *plugin* so it can be integrated by EuGene. Figure 4.4 enhance some of the methods that a *plugin* should implement. Relatively to performance, the most relevant method is the

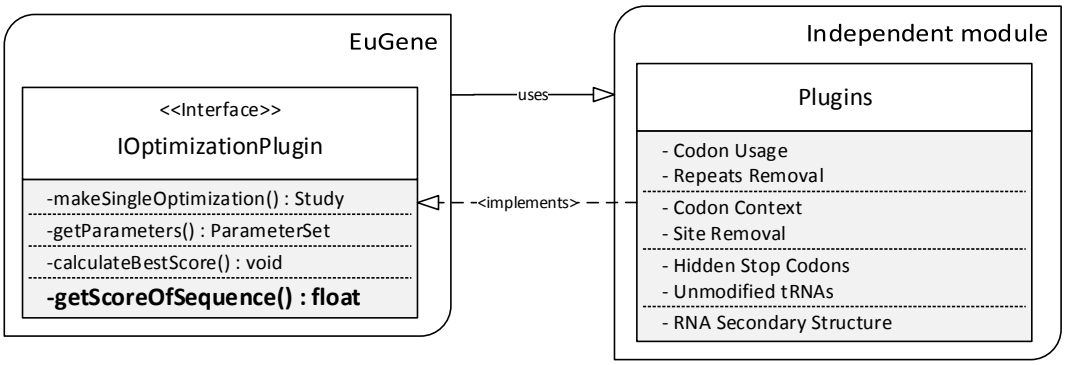


Figure 4.4: Interface that a plugin must respect in order to be used in EuGene

getScoreOfSequence. Its objective is to evaluate a given input codon sequence and return a score from 0 to 100 where 100 is the optimum score. This result is obtained by checking how much the input sequence matches the *plugin* best possible score. Hence, it is noticeable that all the effort done by the plugins occur in this method since it is the one who will evaluate every new input sequence provided by the evolutionary algorithm.

Therefore it is now possible to understand the importance of this method and why it needs to be improved, even in a slighter way. The next subsections show what modification were done at each gene redesign method individually that can toggle their performance.

4.4.1 Codon Usage

The *Codon Usage plugin* has a set of distinct parameters like maximize or minimize the codon usage for a given gene. Also, it allows the gene customization using RSCU (Relative Synonymous Codon Usage) and CAI (Codon Adaptation Index) approaches. The main goal is to count the number of times each of the 64 distinct codons appear using one of the previous approaches. For instance, if the chosen parameter is "Maximize", than this plugin objective is to replace each codon in the gene sequence by the synonymous one that appears often in the whole genome.

The bottleneck of each *plugin* relies in the *getScoreOfSequence* method, and hence, figure 4.5 shows a simple version of the algorithm used by this plugin.

This algorithm is quite simple, where a unique iteration over all the codon sequence is enough to get the codon usage value. However, the *get* methods used to obtain the RSCU

```
...
switch (usageType) {
 // 0 – RSCU
 case 0:
    totalScore += OptimizationRunner.getSelectedHost().getCodonRSCU(sequence.substring(i, i + 3));
    break;
 //1 – CAI
 case 1:
    UsageAndContextTables uct = OptimizationRunner.getSelectedHost().getHouseKeepingGenes().getUsageAndContextTables();
    float tmp = uct.getCodonRelativeAdaptiveness(sequence.substring(i, i + 3));
    totalScore += tmp;
    break;
 default:
    totalScore += 0;
    break;
}
...
```

Figure 4.5: Codon Usage algorithm sample

(*getCodonRSCU*) and CAI (*getCodonRelativeAdaptiveness*) values of each codon can be tricky and present an over complexity that may decrease the algorithm performance.

Let us start with *getCodonRSCU* analysis. This method uses a pre-calculated codon usage table that is calculated when a genome is loaded. Its data is stored in a HashMap structure that hold the list of all 64 codons and the number of occurrences they appear in the genome. Another data that this method uses, is the number of amino acids, present in the genome, as well as the number of synonymous codons for a given codon. All this data is stored in two distinct HashMaps that are also pre-calculated when a genome is loaded. Hence the codon usage value, using RSCU, its given by the following expression:

$$codonUsage = \frac{codonNumberOfOcurrs}{\frac{aaOccurs}{numberOfSynonymous}}$$

This data manipulation is calculated in each iteration of the algorithm presented. Math operations takes CPU time and in this case can be avoided. One simple solution is to pre-calculate these values for all possible codons and stored them in memory. This allows a better access time to get the codon usage of a codon. Hence, the solution adopted was to store the results in an HashMap data structure, which is filled when the genome is loaded. This allows avoiding several unnecessary calls and math operations to get the codon RSCU value for a given codon. As a result, three accesses are avoided:

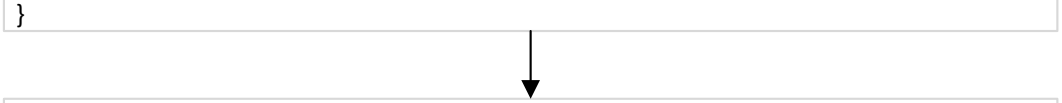
- getting the number of codon occurrences;
- getting the number of amino acids;
- getting the number of synonymous;

Also math operations are avoided and the result is a single access to an HashMap whose implementation provides constant-time performance for the basic operations (*get* and *put*), assuming the hash function disperses the elements properly [16]. Figure 4.6 shows the modifications done.

The method *getCodonRelativeAdaptiveness* uses the same approach as the RSCU used. It calculate the CAI for each given codon every time the methods is called. Hence, in order to

```
public synchronized float getCodonUsageRSCU(String codon)
{
    int numOcc = getCodonUsageCount(codon);
    float aaCount = getAminoAcidCount(genome.getAminoAcidFromCodon(codon));
    float numSyns = genome.getGeneticCodeTable().getNumberOfSynonymous(codon);

    return numOcc / (aaCount / numSyns);
}
```



```
public synchronized float getCodonUsageRSCU(String codon)
{
    return codonUsageRSCUMap.get(codon);
}
```

Figure 4.6: Changes done in *getCodonUsageRSCU* method in order to avoid excessive computation and increase codon usage plugin performance

avoid this overhead, the same strategy was applied. All data that would need to be calculated for a given codon is pre-calculated when the genome is loaded. This data is also stored in an HashMap structure providing the ability to give a codon as key and retrieve the pre-calculated CAI value, also with constant-time performance. As done before, figure 4.7 details the modifications done.

```
public synchronized float getCodonRelativeAdaptiveness(String codon)
{
    Vector<String> syn = genome.getGeneticCodeTable().getSynonymousFromCodon(codon);
    int maxIndex = 0;
    for (int i=1; i < syn.size(); i++)
      if (getCodonUsageFrequency(syn.get(i)) > getCodonUsageFrequency(syn.get(maxIndex)))
        maxIndex = i;

    return getCodonUsageFrequency(codon) / getCodonUsageFrequency(syn.get(maxIndex));
}
```

```
public synchronized float getCodonRelativeAdaptiveness(String codon)
{
    return codonRelativeAdaptivenessMap.get(codon);
}
```

Figure 4.7: Changes done in *getCodonRelativeAdaptiveness* method to avoid excessive computation and increase codon usage plugin performance

4.4.2 Repeats Removal

Some genes have repeated nucleotides or codons chains that may need to be avoided due to translation shifts. Hence, *Repeats Removal* plugin tries to avoid those repetitions by replacing the repeated nucleotides by one or more synonymous codons. Moreover, this replacement should break the repeated chain. Hence, the algorithm to count those nucleotides repetitions is quite trivial, by comparing each nucleotide with the previous one and, if they match, mark it as a repetition. Therefore, at nucleotide point of view, no change was done to the implemented version of the plugin. However, if we want to find codon repetitions and not nucleotides, a simple modification was performed. The input sequence to be optimized was replaced by an *Array* of *Integers* as performed in Simulated Annealing algorithm (section 4.2). This change brings a performance boost since it is computational easier to compare references from an array than *Strings*. Hence, figure 4.8, shows that modification when counting codon repetitions.

```
public float getScoreOfSequence(Study study, String sequence, ParameterSet parameters)
{
    ...
    int consequentRepeats = 0;
    String lastCodon = sequence.substring(0, 3);
    for(int i=3; i<sequence.length(); i+=3) {
        if (sequence.substring(i, i+3).equals(lastCodon))
            consequentRepeats++;
        else{
            if (consequentRepeats >= cThreshold){
                for (int j=(i-(consequentRepeats+1)*3); j<i; j+=3){
                    repeated.set(j/3, repeated.get(j/3)+2);
                }
            }
            consequentRepeats = 0;
        }
        lastCodon = sequence.substring(i, i+3);
    }
}
```

```
public float getScoreOfSequence(Study study, int[] sequence)
{
    int consequentRepeats = 0;
    int lastCodon = sequence[0];
    for (int i = 0; i < sequence.length; i++) {
        if (sequence[i] == lastCodon) {
            consequentRepeats++;
        } else {
            consequentRepeats = 0;
        }
        if (consequentRepeats >= cThreshold) {
            cRepeats++;
        }
        lastCodon = sequence[i];
    }
}
```

Comparing references is faster then comparing Strings

Figure 4.8: Changes done in *Repeats Removal* plugin when removing repeated codon sequences

Using this approach provides smaller and optimized code that uses only references when comparing objects. On the other hand, it was avoided *strings* comparison (with *equals* method) which is quite inefficient [17].

4.4.3 GC Content

Another interesting gene redesign method is the *GC Content*. The main goal here is to count how many ***Gs*** and ***Cs*** bases, exists in a codon sequence. Figure 4.9 illustrates this behavior.

Figure 4.9: Codon sequence highlighting how GC content is measured: green color indicates all the GC bases presented in the sequence. In this example are found 10 bases that match this redesign method out of the 24 bases that mold the sequence

Hence, the implemented version of a possible algorithm evaluates each amino acid of the chain, and check if it is a *G* or a *C* and, if it is, increment the total number of GC found. This evaluation was done directly comparing each character of the sequence against a 'C' or a 'G'. Moreover, despite the performance of comparing characters, it is also wasted some time to evaluate if the condition is valid or not. Figure 4.10 shows a scratch of this algorithm.

```
for (int i=0; i<codon.length(); i++)
    if ((codon.charAt(i) == 'G') || (codon.charAt(i) == 'C'))
        gcContent++;
```

Figure 4.10: Old GC Content plugin algorithm enhancing the block that can be optimized

Since one of the goals of this thesis is to optimize as much as possible the plugins performance, this specific *plugin* can be optimized in the grey block presented in figure 4.10. Hence a possible solution is to use memory, similarly to what was done in the previous plugins. This is a direct trade of between CPU usage and memory that can greatly improve the performance. The chosen approach was to create a map, with all the 64 possible codons, where all their GC content are pre-calculated. This results in a table where for a given key, codon, the correspondent number of GCs is returned. Thus, as before, the performance is vastly improved since it will not be necessary to evaluate each codon over and over again to check its GC number.

This solution was developed using the algorithm presented in figure 4.11 once, for every codon, and the results of each codon stored in a **static** HashMap that is only created one time. The only effort to get the GC content of each codon, became the access time used to retrieve the value from the HashMap.

```
for (int i = 0; i < sequence.length; i++) {
    totalGCContent += cContentMap.get(sequence[i]);
}
```

Figure 4.11: New version of the GC Content plugin algorithm where the only effort is the access time to get values from the HashMap

4.4.4 Codon Context

Some organisms have preference for specific pairs of codons. Thus, some pairs are more frequent in the genome than others. Hence, the goal of the codon context plugin, is to count the number of times each codon appears followed by every other codon. A pair with higher frequency means that this pair is more often found in the genome than pairs with lower frequency [18].

Codon context can be calculated according to the *Codon-Pair Score* (CPS) formula. Thus, the codon context is given by [19]:

$$CPS = \log\left(\frac{ObservedFrequency}{ExpectedFrequency}\right)$$

where:

$$ObservedFrequency = \ codon\ pair\ count\ for\ the\ given\ codons\ in\ the\ genome$$

and,

$$ExpectedFrequency = \frac{(codon1_{frequency} * codon2_{frequency}) * aminoAcidPair_{frequency}}{aminoAcid1_{frequency} * aminoAcid2_{frequency}}$$

Without getting in deep details, since it goes outline of this thesis, these are the operations that codon context plugin needs to do. In the implemented version, these operations are all done every time the *plugin* runs and for each pair of codons present in the input codon sequence. This means that for every codon pair, operations are done repeatedly resulting in poor performance and wasted CPU cycles.

Since none of these parameters are directly related to the input sequence, the codon context for every possible codon pair can be definitely pre-calculated and stored in memory. Once again, this option relieves CPU effort in cost of memory, but brings greater performance. In this case, the codon context value must be calculated for all possible codon pairs, and hence, since there are 64 codons, it is needed to calculate the codon context for $64 * 64 = 4096$ codon pairs. Note that this creates some ambiguous situation, since for instance, if the first codon is given by AAA and the second one is also AAA, it will be calculated two times. However, such effort does not present a major impact.

The structure used to hold this data was, once again, *Maps*. In this particular case, there are two input variables, first and second codon, and therefore a result value associated to this pair. Hence, the need to have two keys to get a single value is perceptive. The solution to

overcome this detail was to have an HashMap structure inside other HashMap as presented next:

Map<Integer, Map<Integer, Float>>

Using this solution, the *key* of the first Map represents first input codon and the *value* is the second Map. The second one has as *key* the second codon and as *value* the codon context associated to the codon pair. Note that this strategy already use as keys integer values that represent each codons as detailed in section 4.2. Hence to get the codon context for a codon pair is as simple as:

codonContext.get(codon1).get(codon2)

As a final result, the final algorithm used to calculate the codon context for a given codon sequence is presented in figure 4.12.

```
float sequenceContext = 0;
for(int i = 0; i < sequence.length - 1; i++){
      sequenceContext += getCodonPairScore.get(sequence[i]).get(sequence[i+1]);
}
```

Figure 4.12: New version of the Codon Context plugin algorithm where the only effort is the access time to get values from the HashMaps

4.4.5 Site Removal

The *Repeat Removal* plugin, sub-section 4.4.2, showed that some sequences needs to be avoided. Therefore, this plugin tackle other specific nucleotides sequences that can be removed due to their deleterious nature. It is important to note that this sequences are processed as nucleotides sequences, meaning that the boundaries of two codons may create a sequence that needs to be avoided. Hence, this *plugin* tackle this problem evaluating the sequence as a nucleotide chain, instead of perform operations over codons. Three distinct sequences that this plugin can avoid are the *Kozak*, *Shine-Dalgarno* and *MazF* sequences:

- Kozak: ***GCCACCAUGG***
- Shine-Dalgarno: ***CCUCCA***
- MazF: ***ACA***

Previously it was said that this algorithm performs the search at nucleotide chain level. Figure 4.13 shows how the edges of the codon *GAA*, marked as yellow, contribute to two distinct sequences that need to be avoided. Thus, the strategy used to identify each codon with a unique integer identifier can not be used.

On the other hand, a similar approach can be used by using a unique integer identifier to each base (*A, C, U and G*). This provides the ability to perform compares using *integers* instead of *string* that is known to be faster. Hence, the input codon sequence used by the ***getScoreOfSequence*** method is converted to the correspondent nucleotide sequence. The trade-off of this solution is that every input sequence needs to be converted into the nucleotide sequence. However, this effort is computational faster to do, since the correspondent amino acid sequence of each codon is pre-calculated and, therefore, the only effort results in the access

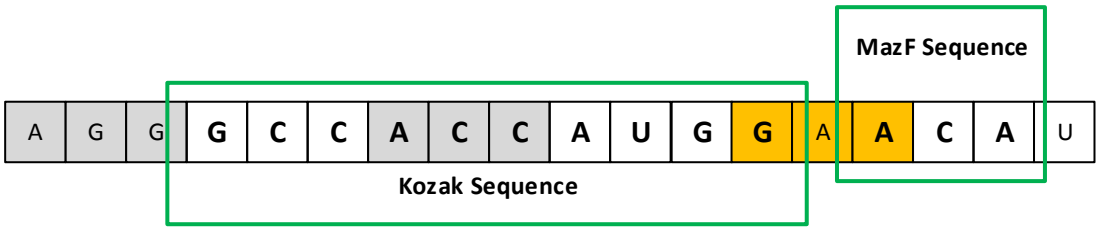


Figure 4.13: Site Removal : sequences that can be avoided. This sequences can start or end at the edges of a codon as represented with yellow. This example shows two distinct sequences, Kozak and MazF and how they can appear in the codon sequence chain.

---

time to the *HashMap* structure where the codon-nucleotides information is stored. Thus, a sequence that does not match exactly the Shine-Dalgarno sequence, might be considered for removal if its hydrogen score, compared with Shine-Dalgarno sequence, is high enough. Hence, when comparing a nucleotide sequence with the Shine-Dalgarno sequence, every base is compared directly with each other, and a specific score is given: the pair **C - G** as score 3, **A - U** 2 and **G - U** 2. If the compare are not done between any of this pairs the given score is 0. Hence, if the total score is higher than a threshold the sequence must be avoided.

The original algorithm of this plugin, calculates each hydrogen bond score every time it advanced a nucleotide in the input sequence as shown in figure 4.14

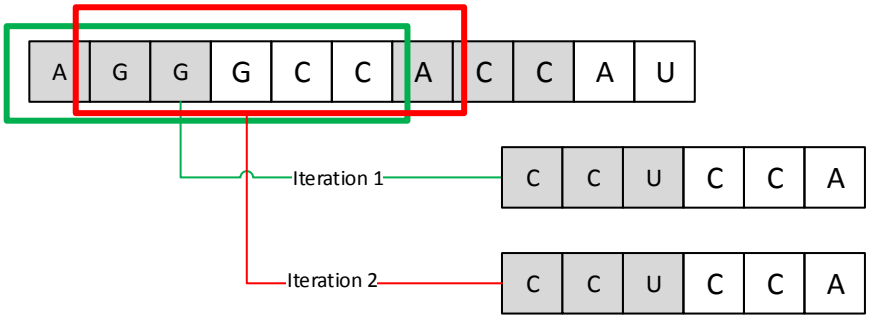


Figure 4.14: Site Removal : iterations effort. At every iteration a hydrogen bound score must be calculated by comparing each amino acid from the input sequence with the specific chain. This example uses the Shine-Dalgarmo Anti Sequence as the String to be compared against

Hence, at each iteration, using the example above, six character compares must be done. The overcome this, it was used, once again, the approach of storing all hydrogen bound scores in memory, being the only computational effort the access time to the structures that hold those values. The structure adopted was similar to the one used in sub-section 4.4.4, two *HashMaps*. One holding as key the first nucleotide input and as value the second *HashMap* that contains as key the second nucleotide and as value the hydrogen bond score associated to the two nucleotides (keys).

4.4.6 Hidden Stop Codons

There are some specific codons known as **stop codons**, that indicate where the translation process should stop. However, many stop codons appear out-of-frame (hidden stop codon).

Those stop codons might show shifted by one or two nucleotide positions leading to a potential decreasing energy and resource waste on nonfunctional proteins [20].

This plugin tackle the issue by placing stop codons in out-of-frame positions through the sequence, so that any miss-translation would stop as soon as possible. To achieve this, when an out-of-frame stop codon is found, the two codons that form it should be replaced by synonymous one. Note that this out-of-frame affect always two codons, meaning that this replacement should be done to codon pairs.

The original implementation of this plugin, tries to find all hidden stop codons by analyzing each pair of codons and check if the full sequence formed by them includes a stop codon starting in any position besides the first. Hence, if a stop codon starts in the second or third character of the pair sequence it is considered an out-of-frame stop codon. Moreover, assuming the codon pair **GAU** - **GAC** the original implementation had several performance issues as shown in figure 4.15.

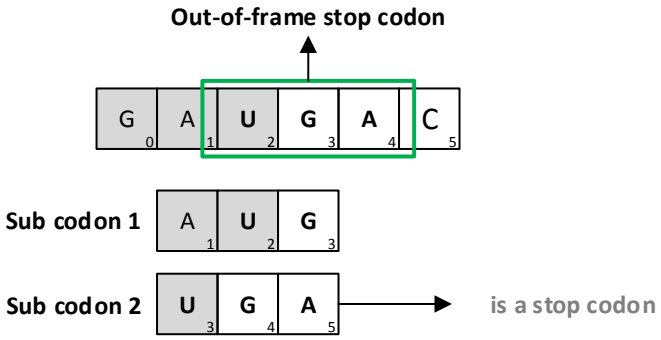


Figure 4.15: Out-of-Frame stop codon - Solution to overcome this issue: Two temporary "codons" must be created losing processing time with the Java operations *substring* and *concat*. After this temporary codons are created, it must be checked if they are a stop codon by comparing strings. This solution is not efficient.

Hence, performing such operations for every input sequence and for every codon pair provided by the simulated annealing algorithm is an exhaustive work that can take a lot of time.

One possible solution to avoid such operations is to calculate every codon-pair possible and evaluate if those pairs have a hidden stop codon. Therefore, a static table can be created to store all codon-pair combinations and the number of hidden stop codons each pair has. This table is created once and thus, the computational effort required to find the hidden stop codons is the access time to that table (table 4.3).

Codon 1	**Codon 2**	**Number Hidden Stop Codons**
AUG	GAC	0
GAU	GAC	1
UUU	CAA	0
AUG	AAA	1
...	...	...

Table 4.3: Example of the Hidden Stop Codon table strategy used by Hidden Stop Codons plugin

To store such data it was used, once again, static *HashMaps* due to their short access

times, and because their elements are only calculated once.

4.4.7 UnModified tRNAs

Along with *Site Removal plugin* there are some specific set of codons that may be avoided. This specific set of codons is different for Bacteria and Eukaryote according to the following:

- Bacteria: ***ACC***, ***CCC***, ***GCC***, ***GUC*** and ***CGG***
- EuKaryote: ***ACG***, ***CCG***, ***GCG***, ***GUG***, ***UCG*** and ***CGG***

Thus, whenever one of this codons is found, if using the plugin for Bacteria or Eukaryote, it should be replace by a synonymous one. Hence, the algorithm is quite simple by comparing every codon from the input sequence with the ones that need to be avoided. On the other hand, the compare is done using the Java *substring* method to get every codon from the input sequence and compare it with the list of codons to be avoided.

Despite being a simple algorithm it can also be optimized. Regarding the input sequence become an *Array* of integers that identify all codons, and the compare is done between integers, it was created a *List*, containing all the codon *integer IDs* that needs to be avoided. With this approach, the only effort is done by checking if the sequence codon *ID* exists in the previously created list. This should present some performance boost.

4.4.8 RNA Secondary Structure

RNA Secondary Structure plugin was developed to predict the secondary structure of a protein. The biological explanation of what is a secondary structure and how it can be achieved is complex to detail and therefore the optimization performed in this plugin was simply done by optimizing the code of the existent algorithm. Hence, the initial algorithm is divided in two distinct blocks as presented in figure 4.16 and figure 4.17.

```
public synchronized float getPseudoEnergie(String sequence)
{
    int pseudo_energy = 0;
    int seqLen = sequence.length();

    /* Calculate folding from the begining. */
    for (int block_size = 2; block_size <= seqLen/2; block_size++)
      pseudo_energy += attraction(sequence.substring(0,block_size), sequence.substring(block_size, 2*block_size));

    /* Calculate folding from the end. */
    for (int block_size = 2; block_size <= seqLen/2; block_size++)
      pseudo_energy += attraction(sequence.substring(seqLen-block_size, seqLen),
                                  sequence.substring(seqLen-block_size*2, seqLen-block_size));

    return pseudo_energy;
}
```

Figure 4.16: RNA Secondary Structure algorithm - block 1

Figure 4.16 shows that the algorithm makes use of two independent blocks. Therefore, instead of using linear programing a parallel solution was adopted.

The ability of a program to concurrently execute multiple code regions is what make parallel (*threads*) programing different from linear. Moreover, Java provides library support

```
public int attraction (String seq1, String seq2)
{
    int pseudoEnergie = 0;
    int len = seq1.length();

    for (int i=0; i<len; i++)
      if ((seq1.charAt(i) == 'A' && seq2.charAt(len-i-1) == 'U') || (seq1.charAt(i) == 'U' && seq2.charAt(len-i-1) == 'A'))
        pseudoEnergie += 2;
      else if ((seq1.charAt(i) == 'C' && seq2.charAt(len-i-1) == 'G') || (seq1.charAt(i) == 'G' && seq2.charAt(len-i-1) == 'C'))
        pseudoEnergie += 3;
      else if ((seq1.charAt(i) == 'U' && seq2.charAt(len-i-1) == 'G') || (seq1.charAt(i) == 'G' && seq2.charAt(len-i-1) == 'U'))
        pseudoEnergie += 1;

    return pseudoEnergie;
}
```

Figure 4.17: RNA Secondary Structure algorithm - block 2

for threads, making them easy to use and to control their concurrent accesses. Also, each thread has its own stack to make method calls and even store variables.

With the help of threads, each ***for*** block from figure 4.16 was put in a different thread so both can be executed simultaneously. After each block completes, the energy of both just needs to be summed to get the final result. Initial tests to validate this solution make use of a gene with 1500 codon length, and a maximum iterations for evolutionary algorithm of 500. Without threads the execution time was about 72000 milliseconds, while just using threads it decreased to 40000 milliseconds.

Figure 4.17 shows that the codon sequence is only iterated once. However, several *string* manipulations are used to check if the characters of the two sequences matches specific nucleotides pairs. Each combination has an associated value, 3, 2 or 1 depending on the nucleotides pair. To avoid these unnecessary operations the solution was to generate an attraction HashMap, where for each codon-pair possible, the attraction value is pre-calculated. Note that to improve even more the performance, the map was created using codon instead of single nucleotides combinations. This allows to iterate faster over the sequence.

As a final result, the validation tests using *Threads* and the memory accesses to get the attraction of two given codons showed a decrease in execution time of 72000 to 16000 milliseconds for the particular case presented before.

4.4.9 Codon Correlation Effect

The *Codon Correlation Effect* redesign method, was developed already with the optimization in mind. However, its first implementation did not make use of the integer strategy, instead of strings, to identify each codon. The goal of this plugin was described in chapter 2, section 2.3.2 and hence, the algorithm behind its implementation is presented in this section.

Since this plugin tries to find which codons are more often used along the gene for each amino acid, it is necessary to build a table with the amino acid list, and therefore, for each amino acid a sub-table containing the list of codons that code for it. Also, it is necessary to calculate the frequency of each codon in the provided sequence. Table 4.18 illustrates what information it is necessary to build for each amino acid.

After this table completion, the remaining step is to evaluate the frequency of each amino acid codons. If the intent of the redesign method is to maximize the correlation effect, i.e, try

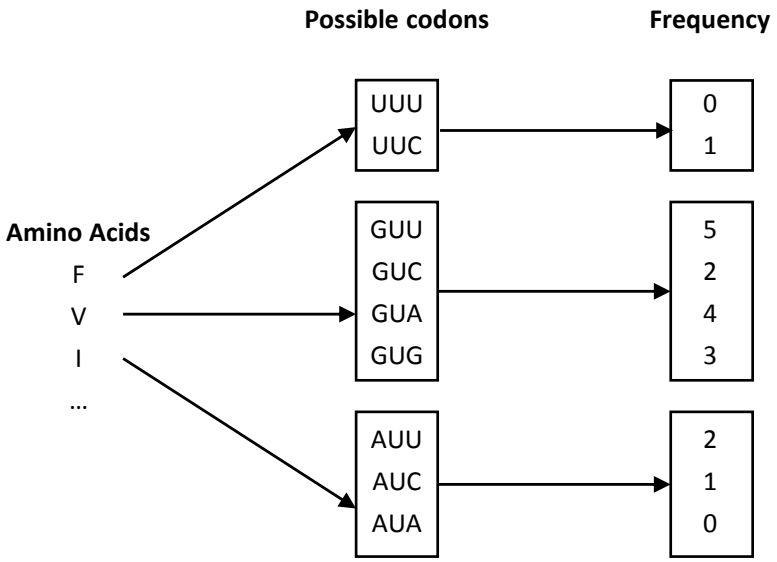


Figure 4.18: Codon Correlation Effect internal table structures

to use always the same codon for each amino acid, the codon with higher frequency for each amino acid is chosen and all others are replaced by them. On the other hand, if the intent is to minimize, it will be used always different codons, as much as possible, for each amino acid. For instance, if for the amino acid F presented in table 4.18, the codon UUU appeared 10 times and codon UUC never appeared in the sequence, every occur of the amino acid F should be replaced by UUU and UUC equally (5 occurrences of each).

4.5 Summary

In this chapter, the optimization system provided by EuGene was analyzed. Moreover, it was found its bottleneck, performance wise, that lead to such investigation. Each redesign method (plugin) code was analyzed and optimized individually using distinct or similar approaches. The main goal of such analysis was to understand the plugins behavior and therefore study methodologies to improve their convergence time (execution time). These methodologies makes use of best practice programming techniques like avoiding unnecessary replication of objects creation, usage of *Threads* to perform parallel computation, use of memory to keep frequently, and final accessed objects. Note that the usage of memory is a trade-off well received between CPU and memory (relieve CPU effort in exchange of memory usage).

After reading this chapter, it should be transmitted all the individual optimizations performed as well as the reasons behind them. Furthermore, it should be clear that good programming techniques leads to optimized, compact and cleaner software code most of the times.

Chapter 5

GEA - Gene Evolution Analysis

The optimizations performed in the previous chapter were tackled with the objective to use faster gene redesign methods. Moreover, exploring all redesign methods parameters to achieve a particular sequence can be computational exhaustive and takes time. Also, to generate all redesign methods parameters as well as random genes, some structures were created. This chapter details the architecture behind be application and how every module contribute to the final outcome.

5.1 Plugins parameters

Having a modular architecture, allowed the redesign methods to be built independently of the main system. Furthermore, the modularity offers several advantages, like the flexibility, that allows the easy expansion of the system, and the independence of a particular module without affecting the main system. Hence, the redesign methods are built separately from the main application, respecting a contract.

This contract, also known as interface in *Java*, says which are the terms that a specific class, that implements this interface, must respect. Furthermore, every redesign method needs to respect the interface *IOptimizationPlugin*. This interface was already developed in EuGene, but the way it was built did not allowed to know what are the exact parameters that a specific redesign method can handle. Furthermore, even if it was possible to know what are the current parameters used in a redesign method, it was also needed to know how they can change. This knowledge is required to build all possible parameters combinations that a redesign method can have.

To overcome this issue, a new entry to the interface was added - *getAvailableParameters*. This method must return the possible parameter list that a redesign method has. Moreover, it is necessary to know what kind of information a parameter carries, for instance what kind of parameter it is - *Boolean, String, Integer of Float* and the range of values it can take. To accomplish this, a new class - *ParameterDetails* - was created. This class has information of a redesign method parameter value and how it can change. This is important to know since later it will be necessary to generate all possible combinations of values for all parameters in all redesign methods. Hence, the *ParameterDetail* class carries the following information:

- Object **Type** - identifies the data type of the parameter (Boolean, Integer, Float or String)

- Object **Value** - the current value of this parameter
- int **minRange** - the minimum value the parameter can take (if the type is Boolean or String this value should be zero)
- int **maxRange** - the maximum value the parameter can take (if the type is Boolean or String this value should be zero)
- boolean **hasCustomValues** - identifies if the parameter has a sub-parameter is a specific value is taken
- Object **whenHasCustomValues** - identifies when the parameter has a sub-parameter

Figure 5.1 illustrates the information that a parameter has. In this particular case, it is possible to use three distinct options, values 1, 2 and 3, in the redesign method presented. Moreover, if the third option is selected it means that it is also necessary to access to an auxiliary structure called *CustomParameterDetails* in order to get the custom value of it and also the range of values that it can have. This strategy allows to create all possible parameters that a plugin can have and also its custom values.

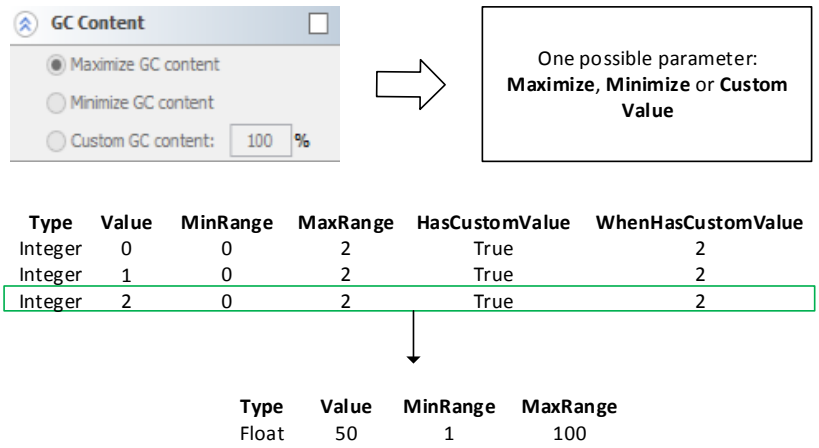


Figure 5.1: Redesign method parameter illustration. Only one option can be active at a time. If a parameter has a custom value, an extra class is needed to accommodate its properties

Hence, every redesign method was updated to respect the new interface. Moreover, using this approach it is possible to generate always all combination of parameters for every plugin easily, even if a new plugin is developed. Furthermore, to generate all possible parameters values of a redesign method becomes easier using this approach. To accomplish the list of all possible parameters, a new class called *PluginParametersList* was created.

In this class, two levels of combinations were created. The first level creates all possible parameters that a redesign method can have as identified in the first table presented in figure 5.1. The second level, shown in the last table of figure 5.1, has all possible combinations for the custom values that may or may not exist for a given parameter. The reason behind this approach is to give equal probability for a parameter to be chosen. In other words, taking the redesign method presented in figure 5.1 as an example, when choosing a parameter to apply to the plugin and process it, it should be equally likely to be chosen Maximize, Minimize and Custom GC Content. Moreover, when one of this is chosen, the

second level is checked and, if it exists, a random value from it is picked up with equal probability.

This strategy allows to generate all possible parameters and choose them with equal probability. Note that all possibilities are generated once and then stored in memory, becoming the only computational effort for the running application, the access to the stored information. The role of the parameter selection is detailed further in section 5.7.

5.2 Random Genes

Another requirement that was presented, was the ability to generate random genes based on a fixed gene - let's call it native gene. Every gene sequence is nothing more than a codon chain that encodes for specific amino acids. Thus, it is possible to change the gene sequence without modifying the resulting protein, as presented in chapter 2, by doing codon substitutions.

Hence, to evaluate how a gene may have evolved, it is necessary to generate random synonymous sequences and study them. To generate a random gene, for each codon of the native gene, it is picked up a random synonymous codon from the genetic code table and a new sequence chain is built. Note that the chosen synonymous may be the same since the chosen option is done with equal probability using the *Java* class *Random*. Once the sequence is built it is added to a *List* that will have as many synonymous sequences as desired. Moreover, this list is final, meaning that its contents will never be changed until the evolution process is finished (section 5.7).

Despite the number of synonymous genes is fixed, it should respect a specific rule. The number of generated synonymous should always be a multiple of the number of available cores of the machine where the application is running. Since every gene sequence is processed in a separated *Thread*, it is desirable that every core is always doing some work to maximize the performance of the global system.

Figure 5.2 exemplifies a situation where the number of synonymous genes is not multiple of the number of cores. For instance, if it is used a machine that has a total of **8** cores and the number of genes generated is **30**, it will be required at least four iterations to process all genes (lets assume that they all complete at the same time). But, since each iteration has 8 available cores, a total of **32 cores** will be available until all genes are processed, resulting in at least two idle cores. To minimize the number of idle cores, the strategy adopted is to use multiples of the number of cores to generated the random sequences. Hence, if using for instance 24 random genes, the possibility of having idle cores is reduced.

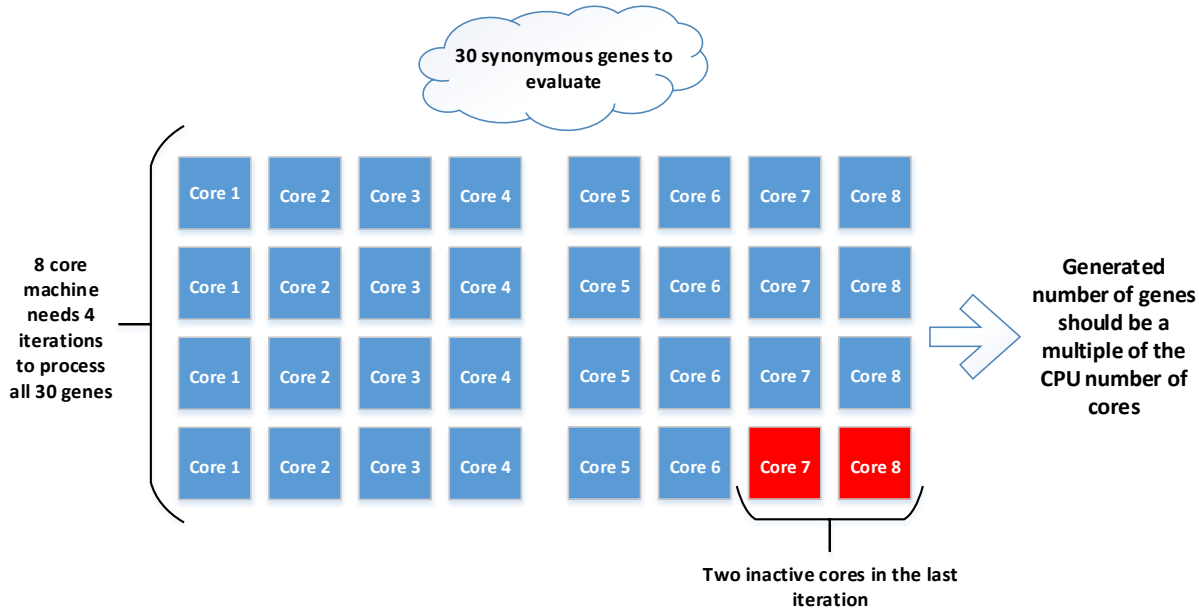


Figure 5.2: This figure illustrate why the number of generated genes should be a multiple of the number of cores of the machine. Not having a multiple number will result in idle cores that could be doing some work and, moreover, delaying the overall system convergence.

5.3 Similarity score

Every time a redesign method is applied to a gene it produces a modified sequence. This sequence represents the optimized chain achieved, using the redesign method purpose and objective. Moreover, one of the requirements of this thesis is to know how similar an optimized sequence is regarding the original gene.

In 1950, Richard Hamming conceived a technique for identification and correction of errors in digital communications [21]. This technique, known as *Hamming Distance*, can be simply defined as the number of bits that are different between two bit vectors. Hence, this technique can be applied to Strings, by checking the number of characters that differs between two words. Figure 5.3 illustrate how the *Hamming Distance* between two strings is calculated.

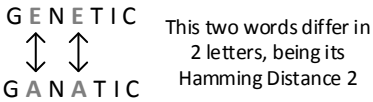


Figure 5.3: Hamming Distance between two words. The number of letters that differ between them defines the Hamming Distance

Since the sequences to be analyzed have always the same size, and the changes between them are always substitutions (one letter to another) and never deletions or insertions, that would lead to different genes, this approach fits to calculate how much the two sequence are similar. On the other hand, if such deletions or insertions exists, a more sophisticated

technique could be used, for instance the *Levenshtein Distance*. The *Levenshtein Distance* is given by the minimum number of deletions, insertions or substitutions required to transform one string into the other [22].

Hence, calculating the similarity score between two strings can be achieved using the Hamming Distance, where the greater the distance is, the more different the strings are. The algorithm to calculate this distance is simple, by just checking the number of characters that differs from two sequences. Once it is known the number of different characters, a similarity score can be assigned. If this number is zero, it means that the sequences are exactly equal and therefore a 100% score is given, on the other hand, if the number of changes is equal to the sequence length a 0% score is assigned. Furthermore, to have a similarity score between 0% and 100% a simple math equation was developed, and implemented, where given the number of changes and the sequence length, it provides the similarity score between the two sequences:

$$score = 100 - ((\frac{hammingDistance}{sequenceLength}) * 100)$$

The output value is given in *float* to improve accuracy.

5.4 Cross-validation

Simulated Annealing finds solutions that may not be optimal as stated in section 4.2. Therefore, for the same set of parameters of a given group of redesign methods, the output sequence can be distinct in some codons and the similarity score achieved may not be accurate. To overcome this issue, and to check that the parameters really improve a random gene to become similar native one, a cross-validation method was developed.

The cross-validation method intends to double-check the veracity of the parameters used. However, despite this strategy adds more complexity to the system (more computational time), tests made showed that without it, the similarity score achieved by a set of parameters may not be consistent. For instance, using the parameters, lets say X, the set of random gene sequences achieved a mean score of 85%. On the other hand, if a new synonymous sequence was generated and the same parameters X were applied the score could be 70%. This huge difference needs to be avoided and to overcome it, a new parameter evaluation is made. This evaluation uses a new set of random genes, four times greater than the previous one. Furthermore, if the initial parameter test was made for 10 gene sequences, when cross-validating, for the same parameters, 40 new gene sequences will be created and tested. The figure 5.4 illustrate this process.

Note that the four time more gene sequences was a naive solution since it can be too many or too less sequences. A proper solution could be calculate this number based on the gene sequence length. For instance, a hypothetical sequence with 3 codons, and with 6 random genes generated, do not need 24 new gene sequences to validate the parameters veracity since most of the new random genes will be equal. Moreover, this solution can also add unnecessary computational effort in trade of a greater veracity result.

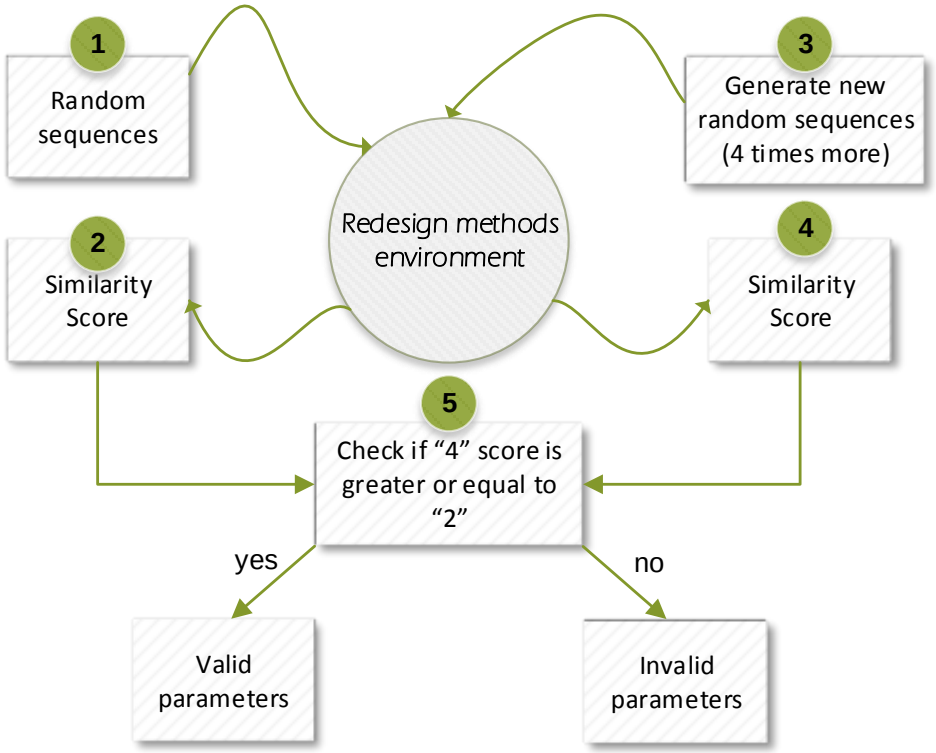


Figure 5.4: Cross-Validation to check redesign methods parameters veracity: numbers identify the process order. First, the fixed random sequences enter the plugin environment system; Second, a similarity score is calculated based on the output sequences provided by the plugins; Third, a new set of random sequences is generated; Fourth, the similarity score is calculated with the objective of checking if the parameters really improve the sequence; Finally if the score of the fourth step is greater or equal to the score of the second step, means that the parameters increase the sequences similarity to the wild type and the parameters may be preserved.

5.5 Plugin weighting

Each gene redesign method has a method called *getScoreOfSequence* that, given an input sequence, evaluates it and assign a score to it, between 0% and 100%, indicating how much the sequence matches the redesign method goal.

Moreover, when using multiple redesign methods simultaneously, the final score can not be given by only one redesign method but by all. Hence, since each redesign method has an individual score, the final score is given by the mean of all scores. This strategy, used in EuGene, provides equal weights to each plugin, which is naive solution since a gene redesign method can have more influence in a protein optimization than another. The solution adopted by EuGene follow the equation:

$$SCORE(m) = \sum^{i} m_i$$

Where the total score of all redesign methods, m, is given by the sum off all individual scores m_i. Hence, to add a weight measure to each redesign method score can by achieved by just multiplying the individual score, m_i, by a weight value, w_i, according to the following:

$$SCORE(m) = \sum^{i} m_i \times w_i$$

The remaining bottleneck is how the weight is calculated. Every time the optimization system calculates a new sequence,it should also provide the weight value for each redesign method. Hence, calculating a new weight is achieved according to the equation:

$$w^* = w + var$$

Thus, w^* represents the new weight to be tested, w the current weight and var is an interval of random values between x and $-x$. Moreover, the x value is calculated at each iteration according to:

$$x = x \times 0.95$$

This solution will make the interval smaller at each iteration, lowering the solution space of possible weights for each redesign method.

5.6 Data storage

Preserving data for future consult or even to create new studies based on that information is vital in every research work. Hence, saving the results achieved by each gene evaluation is necessary for later analysis.

Information can be stored in a computer in several distinct formats. For instance, *XML* (eXtensible Markup Language) is one of the most common formats used since it provides a unique structure that can be easily read by any programing language. However, if one wants to perform data mining over a set of results, there are formats that are more suitable like CSV (Comma-separated values).

CSV file use a special separator character to differentiate different pieces of data. This separator can be a comma, a tab or even a semicolon. Moreover, it allows to store data

in a tabular way where numbers or text are stored in plain-text. Also, every record stored should have the same sequence of fields allowing easy understanding and, more important, data consistence. Every CSV record is expected to have the same structure, being suitable to store the parameters achieved by the evolution system presented in section 5.7.

To store the best parameters achieved for a gene sequence, a new entity called *DataWriter* was created. This class uses the redesign methods parameters information and store it in a CSV file. Also, each record should have all the redesign methods options possible, filled with **0**, if the option is not selected, or **1** otherwise. The figure 5.5 illustrate how the CSV file is filled.

		Site Removal				Codon Context				
Gene Name	Nr Codons	Weight	Shine-Dalg	Kozak	ACA	Weight	Maximize	Minimize	Init Score	Final Score
X	300	0,45	0	0	1	0,93	0	1	75,123	86,532
Y	750	0,05	1	1	0	0,25	1	0	76,2	91,45

Site Removal possible parameters list

Codon Context possible parameters list

Figure 5.5: CSV file format used to store data - this illustration show two redesign mehtods being saved with all the possible parameters of each one. Moreover, the usage of boolean values to indicate which parameter/parameters are selected makes the file easy to read and to make future analyzes.

For instance, it is shown that for *Site Removal* redesign method exists three possible options to be used - Shine-Dalg; Kozak; ACA. When the gene evolution analysis finish, the best parameters achieved for this plugin showed that only the *ACA* option should be used. Furthermore, the initial similarity score should be saved (this score is the mean similarity between all the random genes generated regarding the wild type) to be possible to calculate the improvement achieved with those parameters set. Also, a weight is associated to each redesign method indicating how much the redesign method contribute to the gene improvement - final score. This weight measure was previously described in section 5.5. Finally, the execution time of each gene analysis is also stored (it is not presented in figure 5.5). This time can be useful to estimate how much a evolution takes to finish knowing the number of codons of the sequence used.

5.7 Evolution system

The optimizations performed, along with the new requirements, culminated into an application where all those features were applied and the output stored. Furthermore, this section shows how all pieces, discussed previously, are merged into a single application.

In chapter 3, figure 3.3 showed a simple draft of what is the main goal of this thesis. For a given gene sequence, several synonymous genes are generated and, therefore, a set of gene redesign methods parameters are applied. The intent of this parameters is to make those random genes, as much as possible, similar to the original gene sequence. Furthermore, the individual modules for random genes and redesign parameters generation, similarity calculation, cross-validation and data storage were merged into a system that has as base support the EuGene code. This allows the creation of a new module, to perform such gene evaluation, without reinventing the wheel. Most of the EuGene code was reused, like the genome parsing

tools, internal structures to hold genes information and specially the optimization system, that run, and process, the gene redesign algorithms (plugins). This particular reused code, receives a set of redesign methods (plugins) and run them, through a Simulated Annealing algorithm, that tries to achieve the best output sequence that meet the plugins objective.

Hence, figure 5.6 illustrates an activity diagram describing the whole process that evaluates which plugin parameters contribute to the evolution of a random gene to its current form.

The optimization process runs through a class called *GeneEvolutionRunner*. This class is responsible for getting the gene sequence that is going to be evaluated and the redesign methods that are used in the optimization process. Moreover, it runs an entity called *ParametersSimulatedAnnealing* that for the given input sequence and the list of redesign methods, evolves those methods parameters in order to achieve the best combination possible.

Once the *ParametersSimulatedAnneling* is executed, it is generated a list of all possible parameters for each redesign method used, using the approach detailed in section 5.1. Furthermore, an initial weight is associated to each redesign method. This value needs to be between 0 and 1 (0% and 100%) and the default value was set to *0.5*. Another initialization that needs to be performed is the generation of random genes(section 5.2). This list will have the fixed synonymous sequences, regarding the original gene given as input, that will be evaluated against a set of parameters. Finally, an initial similarity score is measured, comparing each random gene with the original one, using the *Hamming Distance* technique (section 5.3 and, therefore,a mean similarity score is produced. This score acts as the baseline that needs to be improved.

With all the initial steps taken, the redesign method parameters evaluation starts. This evaluation is done through an optimization technique known as *Simulated Annealing*, already described before. Each iteration of this algorithm starts with a random choose of what action is taken from an universe of two possible option - selection of a random plugin to have its parameters changed or selection of a random plugin weight to get its value changed. This approach allows to test both, parameters and weight, with equal probability.

In case of the selected option is to change a parameter, a random plugin is selected (from the list of all possible plugins), and therefore, a random possible parameter for that plugin is chosen from the initial options generated. On the other hand, if the selected option is to change a plugin weight, a random plugin is selected and its weight is modified according to the strategy detailed in section 5.5. Once one of this options is taken, the new set of parameters is ready to be tested.

The plugin execution environment was inherited from EuGene. Hence, for each random gene generated, a new *Thread* is launched, guaranteeing parallelism and distributed effort for each core, since only a number of threads equal to the number of CPU cores is active at a time, with the goal of optimize the gene with the new set of parameters and weights. Once each *Thread* finish its job, it provides a resulting sequence for the set of parameters and weights used. This sequence is then evaluated regarding its similarity against the original sequence. Moreover, the final similarity score is given by the mean score of all the achieved sequences and the original. Hence, it is known how much the new set of parameters improved a set of random genes, and, if this score is greater than the initial one, the set of parameters and weights is saved until score is found. Note that if the score achieved is better than the previous one, a cross-validation is also made, guaranteeing parameters/weights veracity as described in section 5.4.

This algorithm converges once the best achieved score does not change after 10% iterations of the maximum set of iterations fixed. Tests showed that a maximum of 16000 iterations is

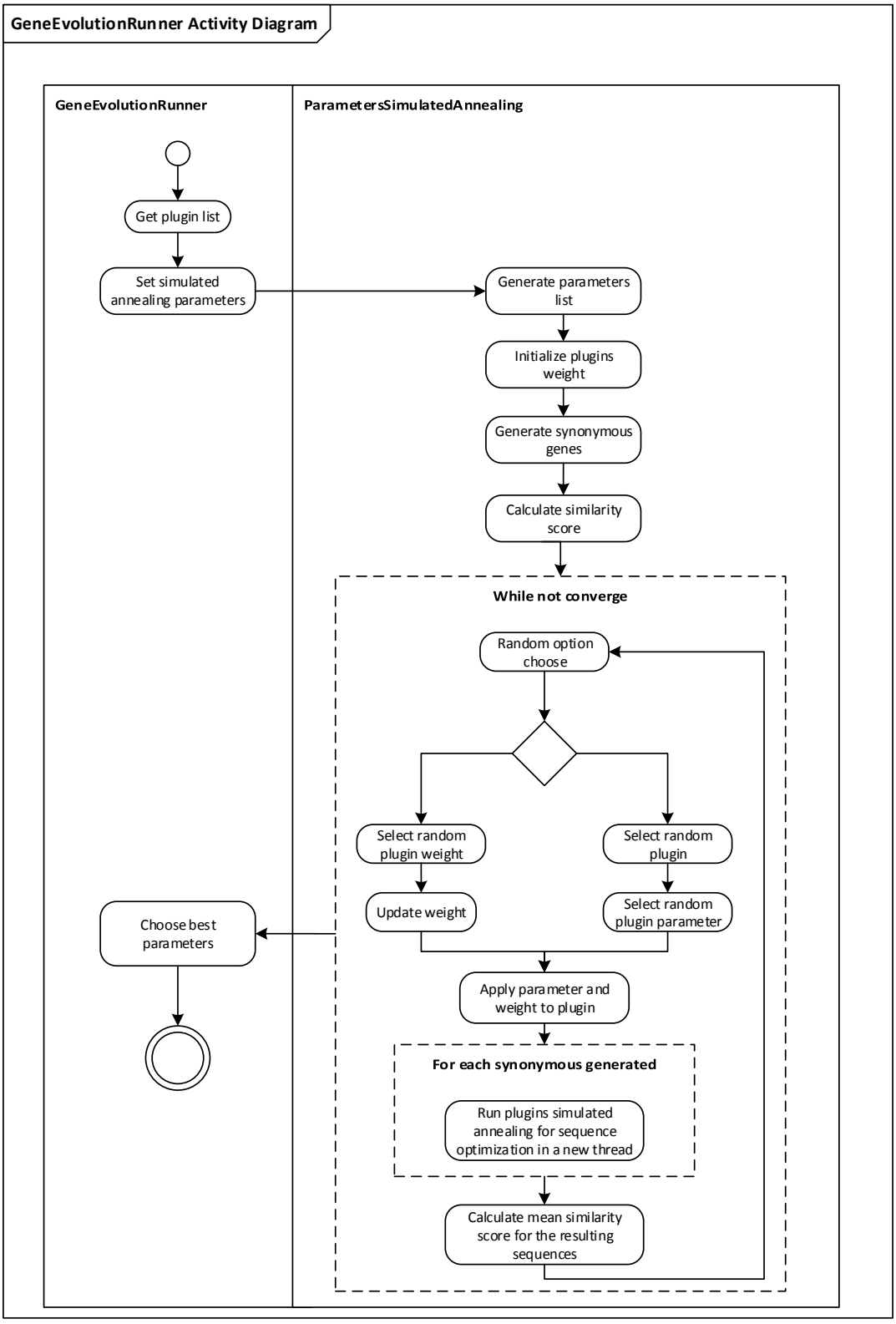


Figure 5.6: Gene evolution activity diagram

enough for the universe of possible parameters generated. This value also brings better guarantee that the achieved parameters are valid, since they need to remain the better parameters found after 1600 iterations without any better combination is found.

5.7.1 User interface

To meet the visual requirements presented in chapter 3, a simple interface was developed, using the same layout as the one presented by EuGene. The screenshot presented in figure 5.7 shows the final appearance of the user interface.

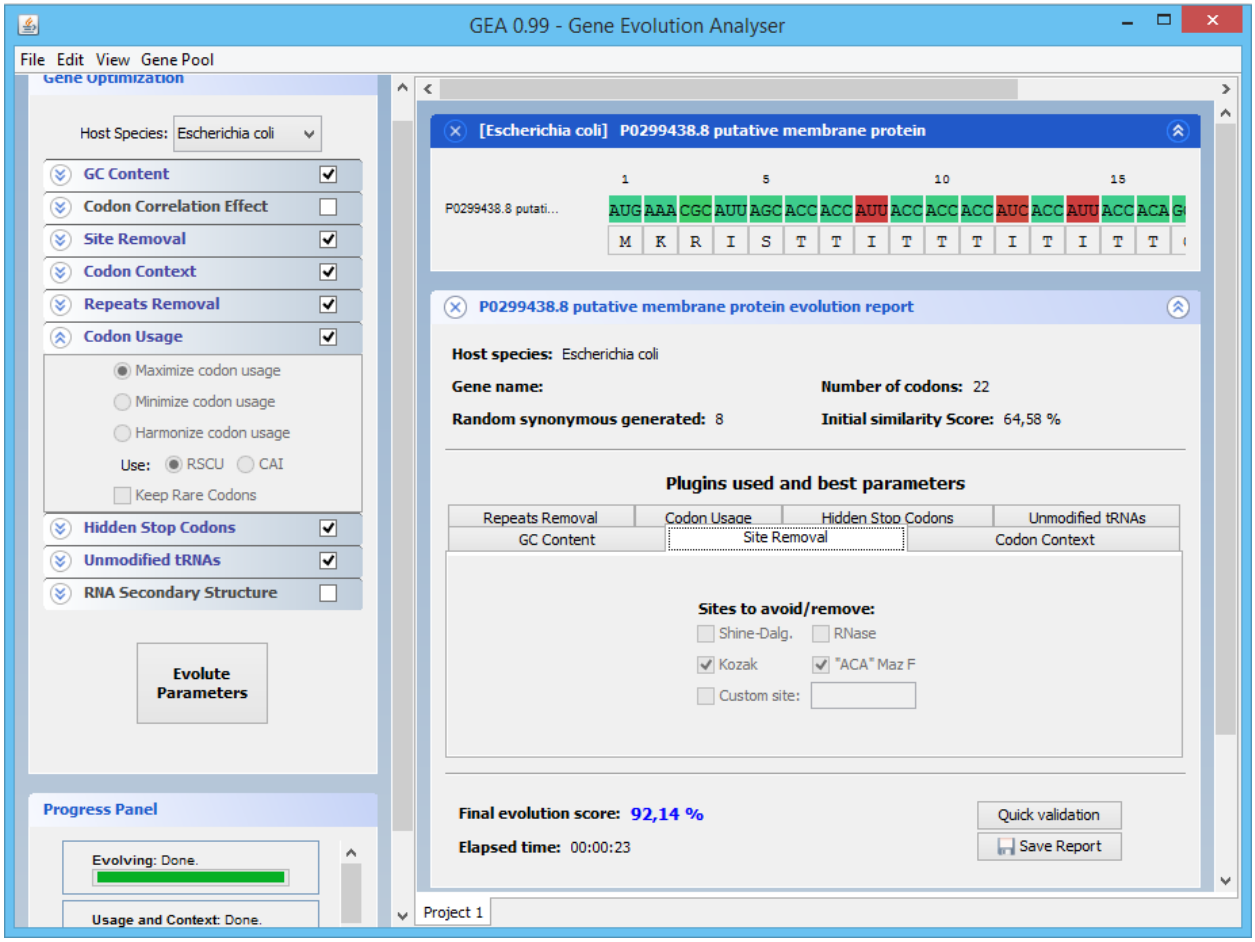


Figure 5.7: GEA application interface

This interface allows to load a genome and evaluate a specific gene choosing the redesign methods desired (left panel). The center panel shows the best values achieved for each redesign method as well as some information about the results achieved, for instance, the initial and final similarity score obtained for a random gene if the set of resulting parameters is used. To validate the parameters veracity, it is also possible to make a cross-validation and define the number of random genes that the user wants the application to test using the parameters achieved. Note that one pendent issue is that the graphical application does not shows the weight used in each plugin.

5.7.2 Automation script

The user interface allows a user to load a genome and selected a gene from it to perform evaluations. However, the main goal, is to evaluate all genes from the genome, and selecting one by one is a tenacious and hard work. This manual selection has a few disadvantages, like the user must be aware that the evolution already finish, then the user must save the resulting data and finally he must select another gene. If a genome has 1000 genes this solution become unpractical for the user.

```
Gene Evolution Runner started...
Evolution analysis started for 8 plugins.

 --- Initiating initial steps ---
N_PROCESSORS: 8
Gene name:      greA

Generating 16 random genes...
Generating list all of possible parameters for selected plugins...
Finished parameters generation...
Codon sequence length:  391
Initial plugins weight: 0.5
Max Global SA iterations:       16000
Max Plugin SA iterations:       6400
Initial similarity score:       77.10997

---- Finished initial steps ----

Now evolving parameters...
Convergence reached 75%... at iteration 1327! Average time of each exterior SA i
teration 15786 achieved score: 85.53%
Elapsed time so far: 20949089

---- Final Results ----
Final score:    85.52725
Number of iterations used:      1727
Average elapsed time for each SA iteration:     15554

GC Content -> 0.47980381843379805
Codon Correlation Effect -> 0.2914716411147012
Site Removal -> 0.6157998966912718
Codon Context -> 0.8527927009638282
Repeats Removal -> 0.06662052673163041
Codon Usage -> 0.6547787372152742
Hidden Stop Codons -> 0.6757366621392534
Unmodified tRNAs -> 0.6943454242076857

Gene Evolution Runner ended.
```

Figure 5.8: GEA application automation script

Hence, a small script was developed to perform analysis to an entire genome. This script analyze each gene from the genome and automatically add the results to an *.csv* file as described in section 5.6. An example of the data stored is presented in appendix A. Note that the analysis for a gene just needs to be performed once, and once a genome is fully analyzed it no longer needs to be evaluated. Figure 5.8 illustrate the developed script running.

5.8 Summary

In this chapter, it was presented the main pieces of the resulting application that converge to the final solution. After reading this chapter it should be possible to understand how each individual component works and how they fit in the GEA application. Moreover, it was presented how the implemented gene redesign methods provided by EuGene needed to be modified, to support custom parametrization from an external source, proving more

information about its contents. This information comes from a new method that all plugins must implement. Also, it was presented how many random genes were generated, enhancing the need of have a number of genes multiple of the number of CPU cores where the application run. Another detailed module is the match between gene sequences where it was detailed the metric used, *Levenshtein Distance* and how the resulting similarity score is obtained. Moreover, the need of having cross-validation to ensure that the redesign methods parameters used fit for the resulting similarity score was presented as well as the usage of a weight measure associated to each redesign method. Finally, the structure adopted to store data was presented, using *.csv* file format, in a way that should be able to perform data mining over the results easily in the future.

Putting together all this modules, the final system was presented as well as an activity diagram that shows how the core of the application works. Also, the resulting interface was presented where it is possible to see the resulting parameters achieved for each gene redesign method. Moreover, it was presented a script that can perform genome analysis without the interaction of the user to choose genes manually.

Chapter 6

Optimization Results

The optimizations performed in chapter 4 resulted in faster plugins convergence and on a faster optimization system. In this section, it is presented the speed improvements achieved by each plugin making a direct comparison between the optimized and the initial versions. Also, some results are presented regarding this thesis main goal, the study of how a set of parameters can improve a random gene to match, as much as possible, its current form.

6.1 Plugins improvement

One off the biggest problems addressed in chapter 4 is the performance presented by each plugin. Moreover, many improvements were performed in order to optimize as much as possible every plugin execution time. Therefore, the following subsections highlight how much the performance was increased in each plugin. The results presented were done under the following conditions:

- Different codon sizes: **50**, **100**, **200**, **400**, **800**, **1000** and **1500**
- Same plugins configuration for each case - *old* and *new* version
- Every plugin was executed **five** times and the average execution time was annotated

These three conditions are enough to known how much the improvement was, and to understand how the modifications performed can vastly improve an algorithm execution time.

6.1.1 Codon Usage

With the optimizations performed in chapter 4, where some math operations were pre-calculated and their values stored in memory, it was possible to avoid excessive computation. Chart 6.1 shows the difference between the old and the optimized version of the *Codon Usage* plugin.

As already said, such tests were made under a set of requirements. Therefore, chart 6.1 shows the execution time of the plugin for 7 distinct cases. Moreover, it is possible to see how the execution time greatly decrease in the optimized version. For instance, when a gene is too small, between 0 and 200 codons, the execution time is really low in both cases, although, despite this fast execution, the difference between times is already considerable (around 4900ms to 1100 ms). In this case, the optimized version of the plugin is about **3.5**

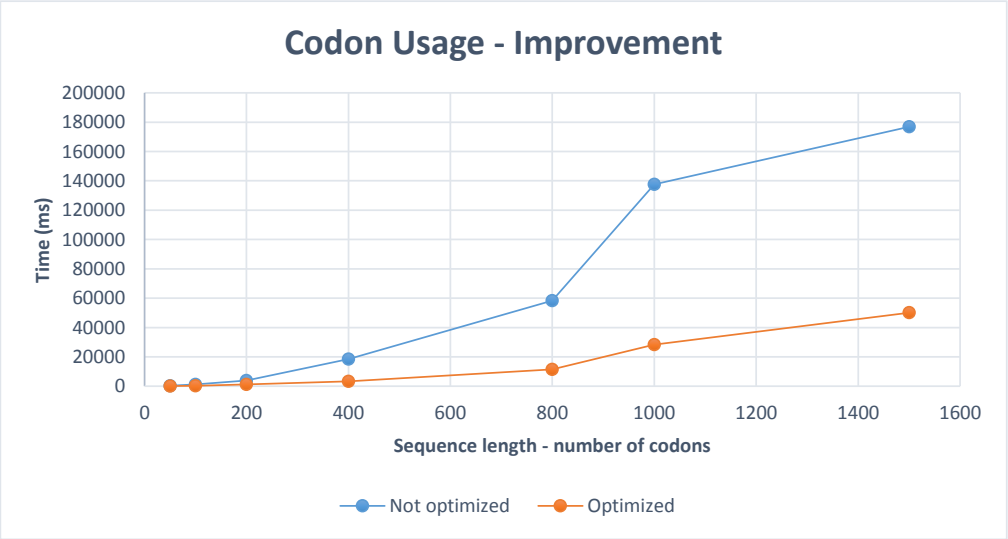


Figure 6.1: *Codon Usage* performance improvement

times faster when optimizing a gene with 200 codons than the ancient version. Hence, when manipulating longer gene sequences, this improvement can lower the execution time from minutes to seconds. In the higher sequence tested, the older version of the plugin took around 180000 milliseconds (3 minutes) and when used the optimized version for the same scenario, the time required to optimize the gene was around 50000 milliseconds (50 seconds).

Thus, when using this plugin repeatedly, in a combinatorial system where it is used several times, this improvement become crucial at the cost of more memory usage. Also, such modification proved to be a win, increasing the plugin performance by about **4.3 times**.

6.1.2 Repeats Removal

Repeats Removal plugin uses a trivial algorithm where a single iteration over the gene sequence is enough to find repetitions. Also, its original execution time was already low but it could still be improved. Chart 6.2 shows the difference between execution times.

By looking at the chart, it is perceptive that in the gene with higher length, the execution time was at most 2500 milliseconds, which is already quite fast. Also, it is noticeable that the optimized version has almost always lower execution time than the ancient version. Note that genes too small are not suitable to grant if the plugin was optimized or not. The main reason for this, is because a gene is so small, that it may have few nucleotides that match the plugin purpose. Also, the computer used, may have more or less cached processes that can improve or decrease the plugin execution time.

In this tests, where the first 2 genes are really short, the execution time was at most 90 milliseconds which is already quite low. However, this optimized version presents an average performance gain of about **1.2** times regarding the ancient version. This also serves as proof that accessing an array of integers and compare its elements, is faster than comparing string. Also, changing array indexes is faster than setting a new string.

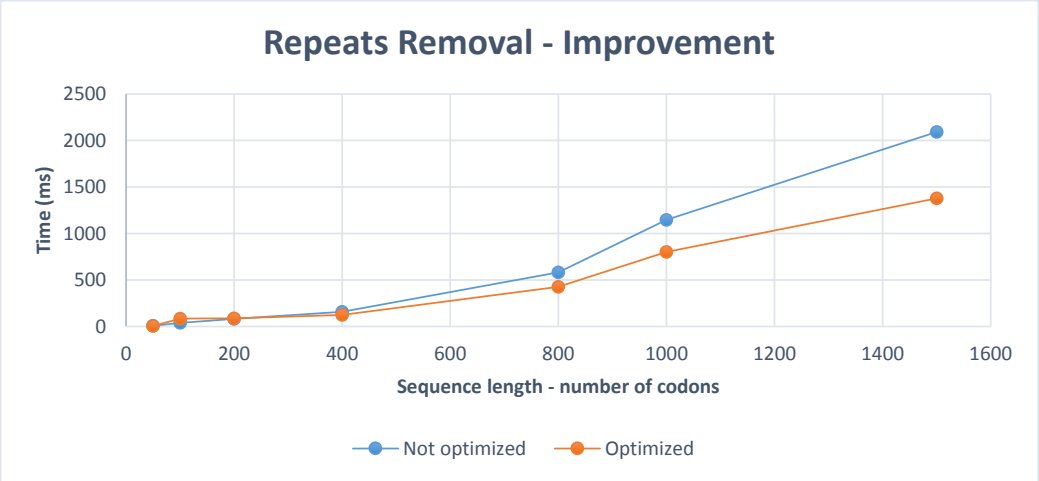


Figure 6.2: *Repeats Removal* performance improvement

6.1.3 GC Content

Along with *Repeats Removal* plugin, *GC Content* plugin also presents low execution times in general, even when using genes with high sequence lengths. Chart 6.3, highlights a direct compare between the holder version of the plugin and the optimized one, where the number of *Gs* and *Cs* were pre-calculated for each possible codon.

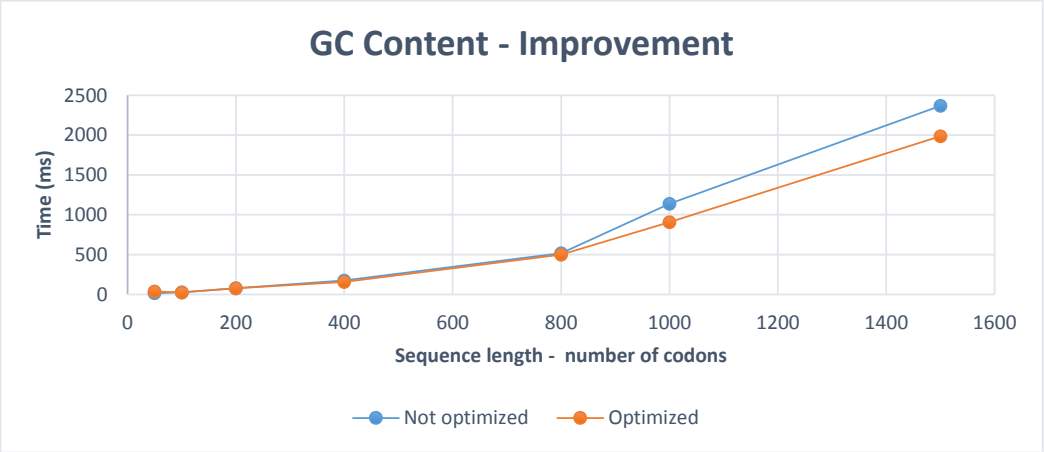


Figure 6.3: *GC Content* performance improvement

It is possible to note that the execution time for the optimized version is a bit lower than the previous version. For instance, in the greater gene, 1500 codons, the ancient version took around **2400 milliseconds** to complete, while the optimized one took **2000 milliseconds**. This difference might not appear to be significant, but, for instance, after 3 gene optimizations, it is possible to achieve the final result **1 second** faster. Moreover, despite the improvement appears minimum, it is a win since the objective of the plugins optimization was to try to lower every millisecond possible in the plugin set.

6.1.4 Codon Context

Codon Context plugin needed to perform a CPS evaluation at each iteration over the codon sequence, as detailed in section 4.4.4. This evaluation uses several mathematical operations that decrease the plugin execution time. However, with the modifications performed, chart 6.4 show the execution time differences between the old and new version of the plugin.

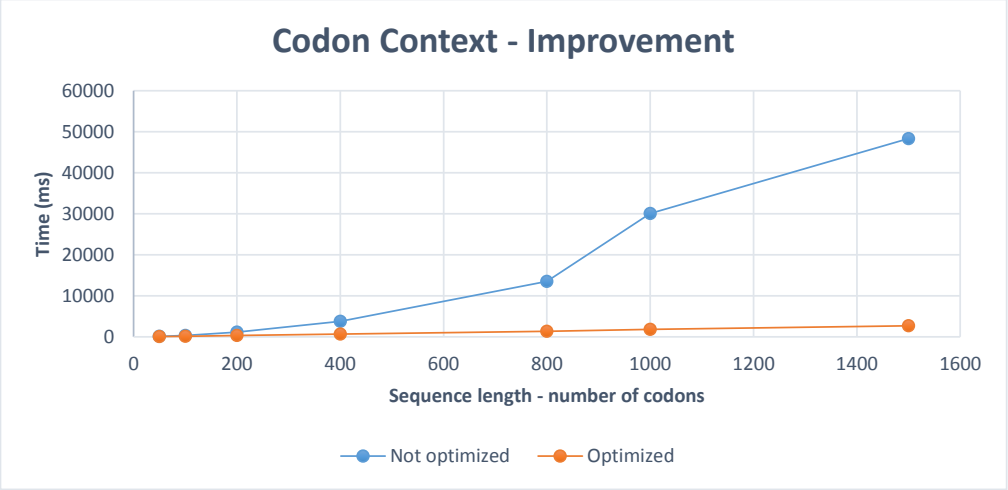


Figure 6.4: *Codon Context* performance improvement

Note that the execution time of the non optimized version greatly increases with the codon length. These times are high, taking in the worst case around 50000 milliseconds to execute. Most of this time is used by the constant mathematical operations that the plugin needs to do. On the other hand, using the strategy of storing all the possible codon-pair combinations and calculating the CPS for each of them, the execution time is greatly decreased.

The chart shows that the optimized version uses almost a constant time to fully execute the plugin. This time is much lower because the effort used by the plugin became just accesses to the *HashTable* structure. Also, those accesses are just retrieval operations, which are fast, being the execution time slightly increase only by the number of codons used (more codons more accesses).

Hence, it is possible to see that the worst case scenario took around 50000 milliseconds to execute against the 2500 of the optimized version. In this particular example, such improvement is about **18 times faster**. As an average time, the results showed an improvement of about 8.4 times. However, this speed up may not be accurate since the low length sequences were used to measure this average and, for instance, genes used with 50 and 100 codons took around 112 and 334 milliseconds respectively in the non optimized version and 64 and 125 milliseconds in optimized version ("only" about 2 times faster). This data is not accurate to make a full measure of the average improvement because the algorithm is not tested exhaustively. Moreover, chart 6.4 shows that the time tends to greatly increase with the codon sequence length in the ancient version and to become "constant" in the optimized version as said previously.

6.1.5 Site Removal

The *Site Removal* plugin addresses the need of avoiding specific nucleotides strings as mentioned in section 4.4.5. With the modifications detailed there, a new evaluation regarding the plugin execution time was done. Chart 6.5 shows a direct comparison between those times and the old and new version of the plugin.

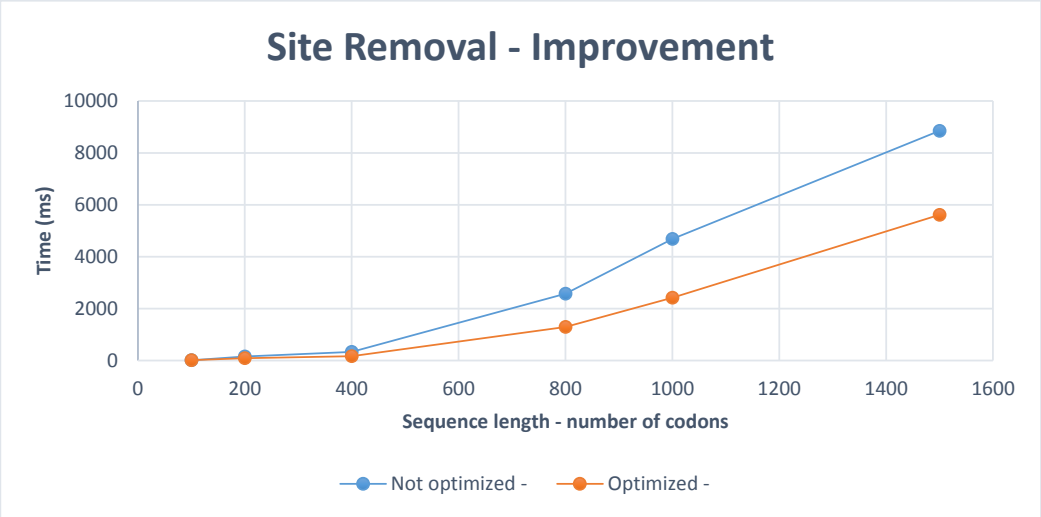


Figure 6.5: *Site Removal* performance improvement

Taking a closer look, it is noticeable that when the codon sequence length is short, the effect of the modifications are not significant. For instance, when using a 200 length gene, the ancient version of the plugin took around 160 milliseconds while the optimized version took 90 milliseconds. On the other hand, larger genes showed that the improvement become more relevant. In the worst case presented, shows a decrease of 3 seconds in the optimized version which is a good result.

Hence, the strategy adopted, storing the hydrogen bond scores in memory, was once again a good approach to achieve better execution times proofing that it is quicker to make accesses to data structures, than making direct comparisons, even with short strings. The average results showed that the improvement was about **1.67** times, which is relevant in an exploratory work, where every second reduced can drastically improve the elapse time of the software.

6.1.6 Hidden Stop Codons

Coming with a solution that could improve the out-of-frame stop codons performance, revealed to bring the best speed up in execution time among all plugins. Thus, the strategy adopted in section 4.4.6 produce the results shown in figure 6.6.

The non optimized version of the plugin has an almost exponential execution time when increasing the number of codons to analyze. This result is explained by the number of operations that were done in each codon-pair. For each, it was necessary to create two additional strings and evaluate each, character by character, to check if an out-of-frame codon

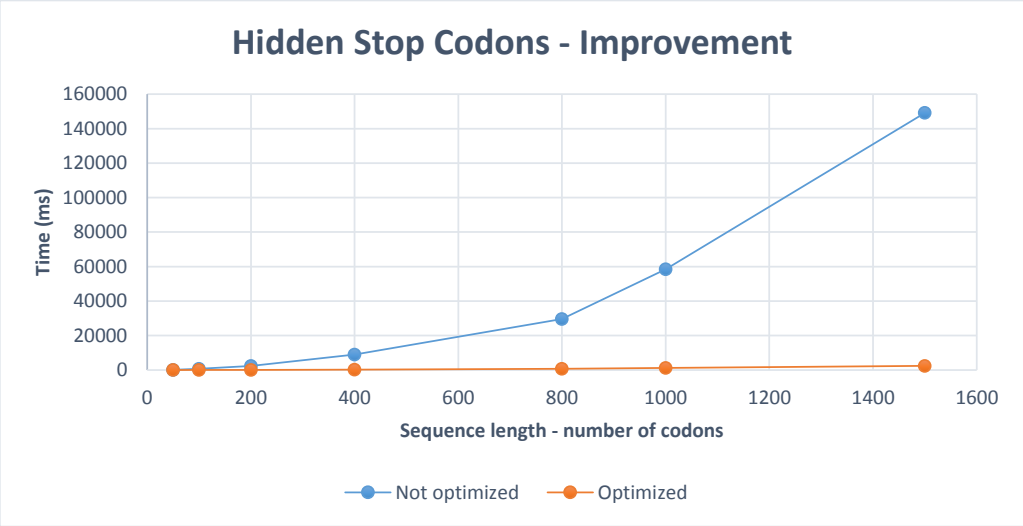


Figure 6.6: *Hidden Stop Codons* performance improvement

was presented. This solution took too much time, and in the worst case studied, it could take up to 160000 milliseconds to achieve the best sequence.

Hence, the solution purposed in section 4.4.6, where every possible out-of-frame was pre-calculated for each codon-pair, saved a lot of time as presented in chart. The execution time decreases drastically, justified by the retrieve operations performance, (*get*), from a map structure for a given codon-pair. Using this approach, the execution time become almost constant since it is not needed to perform any CPU operation in trade of memory accesses.

This plugin clearly exemplifies how faster it is to have objects stored in memory, and access them, than making CPU operations in *runtime*. For instance, the worst case analyzed, using a 1500 codon length, the non optimized version of the plugin took around 160000 milliseconds, as said before, while the optimized version took 2400 milliseconds. This improvement allowed an average speed up of about **43 times faster** than the previous version. In cases where the evaluation could take hours, this improvement allows to achieve the same output result in just a few minutes.

6.1.7 UnModified tRNAs

Most of the improvements achieved in *UnModified tRNAs* plugin result from the usage of Integer *IDs* to identify nucleotides. This change, allows to compare nucleotides using numbers, instead of characters, which is a faster operation. Also, using a List that contains all the nucleotides sequences which needs to be avoided, decreases the compare time, since the compare can be done by checking if the codon to be avoided is present in that list.

Chart 6.7 shows the execution time of the old and new version of the plugin, under the requirements specified in be beginning of the section.

The global execution time of the plugin is already low for a larger gene. However, with the optimization performed, in the worst case, the improvement was about **5 times faster** than the non optimized version. It is also noticeable, that the execution time tends to become more constant when the gene size is increased. This is due the compare between nucleotides

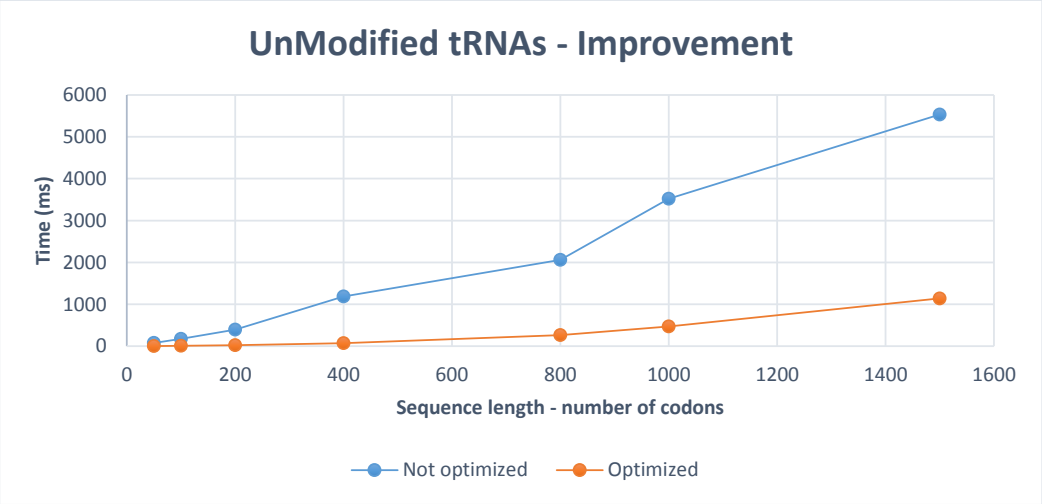


Figure 6.7: *UnModified tRNAs* performance improvement

being done by checking if the codon exists, or not, in the list that contains the sequences to be avoided.

Over time, the usage of this plugin, in an exploratory environment, can also decrease the whole system execution time from even days to just hours.

6.1.8 RNA Secondary Structure

The *RNA Secondary Structure* plugin was the harder plugin to optimize. Due to its nature, it follows a complex algorithm that needs to iterate over several blocks of the gene sequence. Also, besides those iterations, it needs to manipulate strings by making several substrings, for each block, and after that, compare all its characters as explained in section 4.4.8. This implementation requires a lot of CPU effort that takes time. Also, when optimizing a gene for its secondary structure, if the gene has a high sequence length, it can take several minutes for the plugin to end.

Hence, chart 6.8 show the execution time differences between the old and the optimized version of the plugin.

The first thing to notice, is that for this particular plugin, lesser genes were used. Moreover, the genes used had a length of 50, 100, 200 and 400 codons respectively. The reason behind this few test, is the high execution time the plugin takes to achieve the final result. For instance, with a 400 codons length gene, the non optimized version of the plugin takes around 92000 milliseconds to produce the final result. This execution time is already high enough to understand how CPU intensive this plugin is.

With the modifications detailed in section 4.4.8, some of the CPU effort could be relief at the exchange of memory. Also, the usage of *threads* to perform distinct operations simultaneously, allowed the plugin to achieve the final solution faster. Chart shows that in the worst case, where initially the plugin takes around 92000 milliseconds to complete, it now takes around 32000 milliseconds, which is almost **3 times faster**.

This improvement is considerable but is not ideal, since the convergence time is yet high. Also, since the plugin now receives an array of Integer with the codon IDs, a conversion to

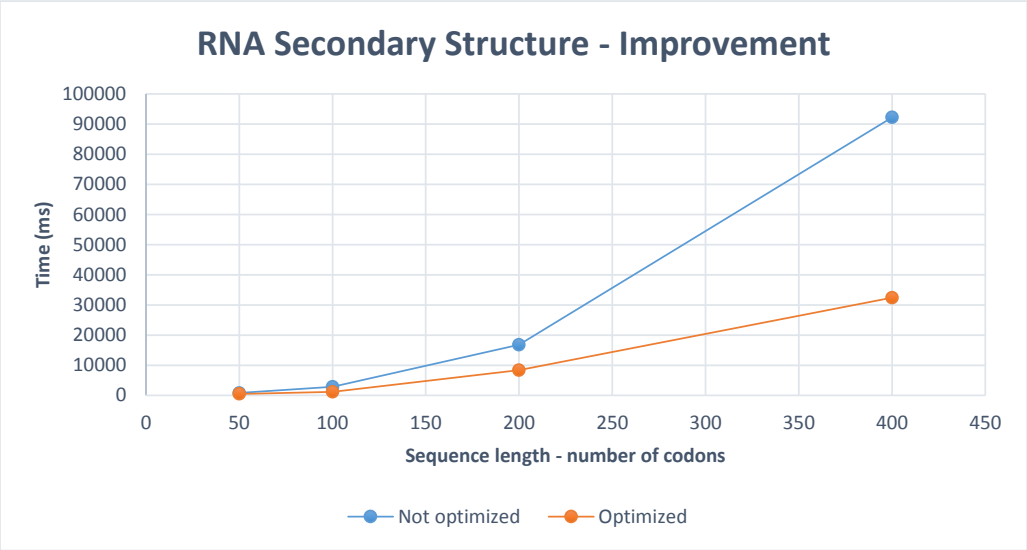


Figure 6.8: *RNA Secondary Structure* performance improvement

nucleotides bases is needed to perform at each plugin call. Note that this needs to be done since the plugin works with nucleotides sequences. The implemented solution allows to get the nucleotide sequence in constant-time (access to HashMap structure) but it is still inefficient since it should not be necessary to perform such conversion every in each plugin call.

However, besides this effort, the plugin still present a considerable speed up against the non optimized version. The final solution implemented shows an average improvement of about **2.2 times**.

6.1.9 Codon Correlation Effect

Codon Correlection Effect redesign method was a new plugin developed with the performance optimization already in mind. However, the first implementation used *strings* to represent each codons and the manipulation performed was over those *strings* to achieve the plugin end, could be slower than if used the strategy of integers to represent codons.

Hence, after making optimizations the algorithm that now uses integers to represent codons, it offered a lower execution time. Chart 6.9 illustrate the average difference between the non optimized version and this newer one.

It is noticeable that the execution time decreased to about the half in the newer version. This is explained by the quicker data manipulation using array indexes instead of the usage of strings to make comparisons. The average results showed that this approach added a speed increase of about **2.6 times**.

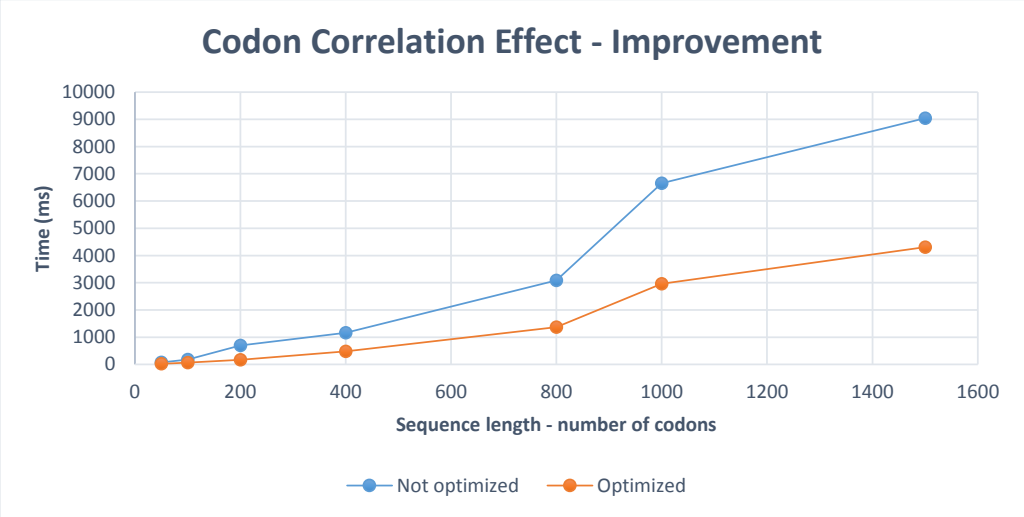


Figure 6.9: *Codon Correlation Effect* performance improvement

6.2 Summary

In this chapter it was presented a direct analysis between the older gene redesign method versions and the optimized ones. Moreover, it was presented the tests made, using different gene sequences lengths, for each case, showing the average improvement achieved. After reading this chapter it should be possible to understand how the modifications performed in chapter 4 culminated into smaller redesign methods execution times, leading to faster convergence. Moreover, table 6.1 shows a concise information regarding these improvements.

	50	100	200	400	800	1000	1500	Average
Codon Usage	2,84	4,91	3,46	5,45	5,11	4,84	3,53	**4,31**
Repeats Removal	1,60	0,46	0,92	1,28	1,37	1,43	1,52	**1,23**
GC Content	0,39	1,04	0,97	1,11	1,04	1,25	1,19	**1,00**
Codon Context	1,75	2,69	3,55	5,62	10,18	16,75	17,98	**8,36**
Site Removal	-	0,67	1,84	2,01	1,98	1,94	1,58	**1,67**
Hidden Stop Codons	22,00	38,81	44,35	48,34	42,14	45,44	61,15	**43,17**
UnModified tRNAs	38,00	25,14	18,76	16,01	7,87	7,46	4,85	**16,87**
RNA Sec. Structure	1,73	2,41	2,01	2,84	-	-	-	**2,25**
Codon Corr. Effect	3,23	2,88	4,15	2,44	2,26	2,26	2,10	**2,76**

Table 6.1: Summary of the average improvements of each plugin: for each codon sequence length, an average speed up is presented. Each value represent how much times the speed was improved concerning the initial plugins state. The last column (Average) represents the mean of all values achieved and, therefore, the global speed up obtained.

Chapter 7

Conclusions

The main goals of this thesis included the implementation of new gene redesign methods, the analysis of the optimization system present in EuGene, and the study of how it can be improved. The speed improvement of known gene redesign methods and their exploitation for another end has also been tackled. In this thesis those methods were used to find what processes a gene went through, until it reached its current state. This was accomplished by performing combinations of gene redesign methods and their parameters.

Most of these targets were implemented directly in EuGene since it has a solid and well defined structure, allowing to test the implemented improvements in each gene redesign method. Moreover, the plugin system used by EuGene to run different redesign methods, allows the creation of new algorithms by loading them from different modules, and hence, the implementation of a new gene redesign method becomes easy to merge once it respects a specific interface. In this work three distinct redesign algorithms were studied and implemented - Ramp Effect, Codon Correlation Effect and Keep Rare Codons, being Ramp Effect and Codon Correlation Effect used in a single plugin because of their similar implementation. Furthermore, new several optimization techniques were studied and applied to each redesign method available, with the goal of improving their execution times. To accomplish this, some techniques like the storage in memory of objects to hold information that is accessed repeatedly, the refactoring of code avoiding unnecessary variables instantiations, the use of an Integer Array to represent a sequence (allowing faster sequences manipulations), among others were applied. On the other hand, an application that can use the gene redesign methods to find how a gene evolve to its current state was developed. This makes uses of all redesign methods presented in EuGene and tries to find the best parameters combination, using optimization algorithms, that can explain how a gene evolve. Note that this evaluation is done once for every gene, and that the execution time can be high due the many combinations that are tested.

Hence, the goals presented in chapter 3 result into a single application that explores the gene redesign methods, using optimized algorithms and techniques. Moreover, the resulting platform automatically evaluate every gene presented in a genome without the need of manually select a gene. This work is an in-progress task until the genome is fully analyzed. The results are stored each time a gene is analyzed so it can be used by researchers anytime without waiting for a genome to be completely evaluated.

7.1 Future Work

Following the goals and requirements of this thesis, there are ideas that require further investigation. For instance, since every gene redesign method now makes use of an Array of Integers to represent a codon sequence, it is now hard to manipulate sequences at nucleotide level, because the input sequence is always a codon sequence. To overcome this, new algorithms for the same gene redesign method can be investigated using codons for the same end.

Despite having improvements in codon manipulation, the evolutionary algorithm can still be improved. There are several variables that can make the algorithm converge like the stagnation of the score, for a given number iterations, and the maximum number of iterations, without convergence, of the algorithm. Some quick tests showed that there is a relationship between the number of codons of a sequence and the maximum number of iterations that the Simulated Annealing can take. For instance, with a maximum number of iterations of 5000, lets assume that the score is 80%. If for the same gene we establish a maximum number of iterations of 12000, the result could be 80.1% and the execution time would be much higher for such a small percentage that does not bring greater impact in the protein expression. Therefore, finding the best relation between the sequence size and the number of iterations of the Simulated Annealing can increase the speed of the redesign methods.

On the other hand, the number of synonymous sequences generated also has a relationship with the codon sequence length. It would be interesting to generate a fixed number of synonymous sequences when finding the best parameters that can explain how a gene evolve. This would reduce the execution time of the gene analysis as well as improve the parameters veracity. For instance it is unnecessary to generate, for a hypothetical gene with 3 codon length, 20 synonymous, since most of them would be repeated and would decrease the analyzes execution time.

Also a biological analysis needs to be performed over the results achieved. The output data structure already provides an easy way to perform data mining over the results and, therefore, some patterns could be found. For instance, it could be observed that a given genome zone was highly affected by the codon usage and other zone was affected by Codon Correlation Effect or Codon Context. This can lead to further investigations and possible explanations regarding gene evolution.

Bibliography

[1] Stryer L. Biochemistry. Berg JM, Tymoczko JL. DNA Illustrates the Relation between Form and Function. In W H Freeman, editor, *Biochemistry*, chapter Section 1. New York, 5 edition, 2002.

[2] Miller JH Griffiths AJF, Gelbart WM. *Modern Genetic Analysis*. W. H. Freeman, New York, 1999.

[3] Tamir Tuller, Asaf Carmi, Kalin Vestsigian, Sivan Navon, Yuval Dorfan, John Zaborske, Tao Pan, Orna Dahan, Itay Furman, and Yitzhak Pilpel. An evolutionarily conserved mechanism for controlling the efficiency of protein translation. *Cell*, 141(2):344–54, April 2010.

[4] Hiroshi Akashi. Translational selection and yeast proteome evolution. *Genetics*, 164(4):1291–303, August 2003.

[5] Gina Cannarozzi, Gina Cannarrozzi, Nicol N Schraudolph, Mahamadou Faty, Peter von Rohr, Markus T Friberg, Alexander C Roth, Pedro Gonnet, Gaston Gonnet, and Yves Barral. A role for codon order in translation dynamics. *Cell*, 141(2):355–67, April 2010.

[6] Evelina Angov, Collette J Hillier, Randall L Kincaid, and Jeffrey a Lyon. Heterologous protein expression is enhanced by harmonizing the codon usage frequencies of the target gene with those of the expression host. *PloS one*, 3(5):e2189, January 2008.

[7] T Ikemura. Codon usage and tRNA content in unicellular and multicellular organisms. *Molecular biology and evolution*, 2(1):13–34, January 1985.

[8] Dequan Chen and Donald E Texada. Low-usage codons and rare codons of Escherichia coli Mini Review. 10(c):1–12, 2006.

[9] H Rüssmann, H Shams, F Poblete, Y Fu, J E Galán, and R O Donis. Delivery of epitopes by the Salmonella type III secretion system for vaccine development. *Science (New York, N.Y.)*, 281(5376):565–8, July 1998.

[10] Paulo Gaspar, José Luís Oliveira, Jörg Frommlet, Manuel a S Santos, and Gabriela Moura. EuGene: maximizing synthetic gene design for heterologous expression. *Bioinformatics (Oxford, England)*, 28(20):2683–4, October 2012.

[11] Jason Brownlee. *Clever Algorithms: Nature-Inspired Programming Recipes*. LuLu, first edit edition, 2011.

[12] Guillaume Cambray and Didier Mazel. Synonymous genes explore different evolutionary landscapes. *PLoS genetics*, 4(11):e1000256, November 2008.

[13] Matan Ninio and Johannes J Schneider. Weight annealing. 349:649–666, 2005.

[14] Yaghout Nourani and Bjarne Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373–8385, October 1998.

[15] Jack Shirazi. *Java Performance Tuning*. O'Reilly, 2nd edition, 2000.

[16] Oracle. Java Platform, Standard Edition 6 API Specification.

[17] Glen Mccluskey, Steve Buroff, and Craig Hondo. Thirty Ways to Improve the Performance of Your Java Programs trademark appears alone it is a reference to the Java Development. (October):1–37, 1999.

[18] Paulo Gaspar. *Gene optimization for heterologous expression*. Master thesis, University of Aveiro, 2010.

[19] J Robert Coleman, Dimitris Papamichail, Steven Skiena, Bruce Futcher, and Steffen Mueller. Virus Attenuation by Genome-Scale Changes in Codon Pair Bias. 320(5884):1784–1787, 2009.

[20] Hervé Seligmann and David D. Pollock. The Ambush Hypothesis: Hidden Stop Codons Prevent Off-Frame Gene Reading. *DNA and Cell Biology*, 23(10):701–705, 2004.

[21] R.W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Tech Journal*, 26:147–160, 1950.

[22] Rishin Haldar and Debajyoti Mukhopadhyay. Levenshtein Distance Technique in Dictionary Lookup Methods : An Improved Approach. (Ld).

Appendix A

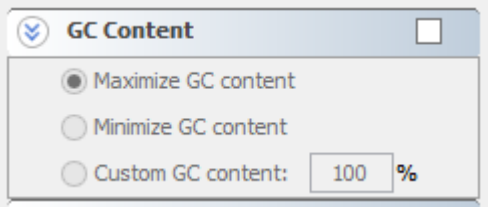
GEA output samples

The following images illustrate how the results achieved by the gene evolution analysis system (GEA) are stored. For each record (gene) is presented what kind of parameters are more suitable to make a synonymous gene become similar to the original one. The final similarity score is calculated using all redesign methods. Note that due to the large number of parameters, the *.csv file* has a lot of columns and, therefore, the stored information had to be split in several illustrations to be presented in this appendix.

Each redesign method has its parameters filled with **1** or **0**, meaning what parameters options should and should not be used to the achieve maximum similarity score. Moreover, the weight associate to each plugin (plugin relevance in the final result) is also presented.

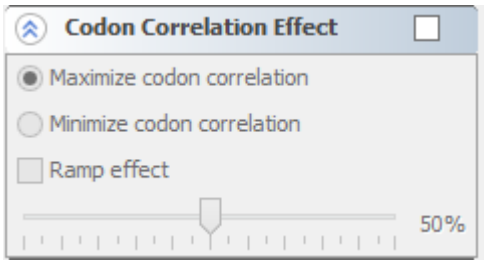
The tests performed do not include the *RNA Secondary Structure* redesign method, since its execution time is yet too high. For instance, a simple gene optimization that took 10 milliseconds (400 length gene sequence), would took around 55 milliseconds if this method was included, according to tests made.

- 3 possible options
- Only one can be selected
- Custom option has a range of [10, 90] possible options



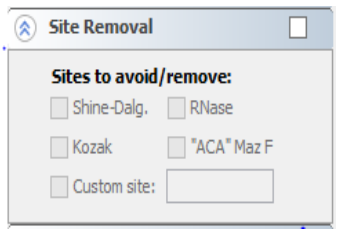
			GCCONTENT1											
Gene Name	**Number of Codons**	**Weight**	**Maximize**	**Minimize**	**Custom**	**10**	**20**	**30**	**40**	**50**	**60**	**70**	**80**	**90**
yggN	240	0,631	0	1	0	0	0	0	0	0	0	0	0	0
yggL	119	0,513	1	0	0	0	0	0	0	0	0	0	0	0
ydfO	142	0,579	1	0	0	0	0	0	0	0	0	0	0	0
b1550	59	0,369	0	0	1	0	0	0	0	0	1	0	0	0
ribB	218	0,417	0	1	0	0	0	0	0	0	0	0	0	0
b3042	117	0,127	1	0	0	0	0	0	0	0	0	0	0	0
b2387	109	0,723	0	1	0	0	0	0	0	0	0	0	0	0
glk	322	0,401	0	1	0	0	0	0	0	0	0	0	0	0
tsx	295	0,641	0	1	0	0	0	0	0	0	0	0	0	0
yajI	200	0,009	1	0	0	0	0	0	0	0	0	0	0	0
b1009	267	0,669	0	1	0	0	0	0	0	0	0	0	0	0
b1010	129	0,954	1	0	0	0	0	0	0	0	0	0	0	0
rpsQ	85	0,475	1	0	0	0	0	0	0	0	0	0	0	0
rpmC	64	0,438	1	0	0	0	0	0	0	0	0	0	0	0
ydhD	116	0,779	1	0	0	0	0	0	0	0	0	0	0	0
sodB	194	0,466	0	1	0	0	0	0	0	0	0	0	0	0
ylcB	458	0,337	0	1	0	0	0	0	0	0	0	0	0	0
ylcC	111	0,421	1	0	0	0	0	0	0	0	0	0	0	0
yjeT	66	0,357	1	0	0	0	0	0	0	0	0	0	0	0
purA	433	0,489	0	0	1	0	0	0	0	0	0	0	0	1
recB	1181	0,374	0	0	1	0	1	0	0	0	0	0	0	0

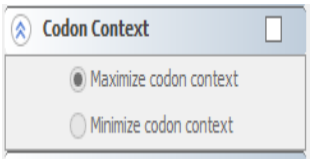
- 3 possible options
- Only Maximize or Minimize can be selected
- Ramp effect is optional and has a range [10,60] of possible values



		CODONCORRELATIONEFFECT1									
Gene Name	**Number of Codons**	**Weight**	**Maximize**	**Minimize**	**Ramp Effect**	**10**	**20**	**30**	**40**	**50**	**60**
yggN	240	9E-05	0	1	0	0	0	0	0	0	0
yggL	119	0,729	1	0	1	0	1	0	0	0	0
ydfO	142	0,926	0	1	1	0	0	0	1	0	0
b1550	59	0,590	1	0	0	0	0	0	0	0	0
ribB	218	0,544	0	1	0	0	0	0	0	0	0
b3042	117	0,000	0	1	0	0	0	0	0	0	0
b2387	109	0,580	1	0	0	0	0	0	0	0	0
glk	322	0,495	0	1	0	0	0	0	0	0	0
tsx	295	0,480	1	0	0	0	0	0	0	0	0
yajI	200	0,943	0	1	0	0	0	0	0	0	0
b1009	267	0,715	0	1	0	0	0	0	0	0	0
b1010	129	0,544	1	0	1	0	0	0	1	0	0
rpsQ	85	0,477	0	1	1	0	0	1	0	0	0
rpmC	64	0,395	0	1	0	0	0	0	0	0	0
ydhD	116	0,860	0	1	1	0	0	0	1	0	0
sodB	194	0,430	1	0	0	0	0	0	0	0	0
ylcB	458	0,634	0	1	0	0	0	0	0	0	0
ylcC	111	0,421	0	1	1	0	0	0	0	1	0
yjeT	66	0,444	0	1	0	0	0	0	0	0	0
purA	433	0,305	0	1	1	0	0	1	0	0	0
recB	1181	0,460	1	0	0	0	0	0	0	0	0

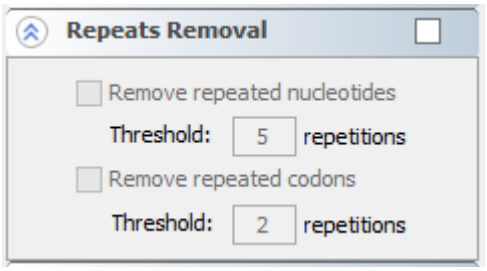
- Site Removal has all options optional – For tests only 3 options were used
- Codon Context has two possible options, only one can be selected





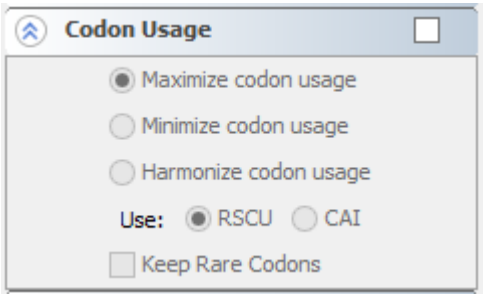
		SITEREMOVAL1				CODONCONTEXT1		
Gene Name	**Number of Codons**	**Weight**	**Shine-Dalg**	**Kozak**	**ACA**	**Weight**	**Maximize**	**Minimize**
yggN	240	0,115	0	1	1	0,350	0	1
yggL	119	0,783	1	0	0	0,579	1	0
ydfO	142	0,500	1	1	0	0,949	0	1
b1550	59	0,528	0	0	0	0,532	0	1
ribB	218	0,001	1	1	1	0,182	0	1
b3042	117	0,510	0	0	1	0,504	0	1
b2387	109	0,625	0	0	1	0,786	0	1
glk	322	0,420	1	1	0	0,507	0	1
tsx	295	0,535	0	0	1	0,603	1	0
yajI	200	0,884	0	0	1	0,293	0	1
b1009	267	0,743	1	0	1	0,728	0	1
b1010	129	0,146	1	1	1	0,534	1	0
rpsQ	85	0,197	1	1	0	0,582	0	1
rpmC	64	0,756	0	1	0	0,000	0	1
ydhD	116	0,885	0	1	1	0,101	0	1
sodB	194	0,604	1	0	0	0,528	0	1
ylcB	458	0,796	0	0	0	0,849	1	0
ylcC	111	0,221	0	1	1	0,668	1	0
yjeT	66	0,975	0	1	0	0,523	1	0
purA	433	0,729	1	0	1	0,339	1	0
recB	1181	0,002	0	1	0	0,254	0	1

- All options are optional
- Each selected option has a set of possible values: [3, 12] for nucleotide repetitions and [2,5] for codon repetitions



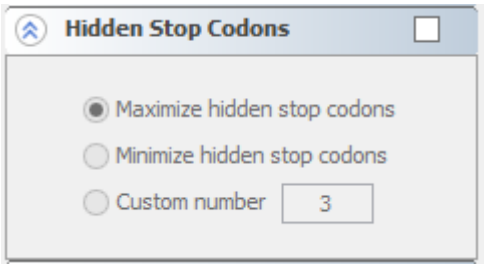
			REPEATSREMOVAL1														
Gene Name	**Number of Codons**	**Weight**	**RR Nucleotides**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**RR Codons**	**2**	**3**	**4**	**5**
yggN	240	0,709	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
yggL	119	0,869	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ydfO	142	0,230	1	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0
b1550	59	0,381	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ribB	218	0,054	1	0	0	0	1	0	0	0	0	0	0	1	0	1	0	0
b3042	117	0,244	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
b2387	109	0,575	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
glk	322	0,330	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tsx	295	0,763	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
yajI	200	0,387	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b1009	267	0,155	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
b1010	129	0,676	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rpsQ	85	0,280	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
rpmC	64	0,516	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
ydhD	116	0,618	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
sodB	194	0,285	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ylcB	458	0,674	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
ylcC	111	0,367	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
yjeT	66	0,858	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
purA	433	0,292	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
recB	1181	0,502	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0

- 6 possible options
- Only Maximize, Minimize or Harmonize can be selected at a time
- If harmonize is selected RSCU or CAI must be used
- Every case has the possibility to keep rare codons options selected



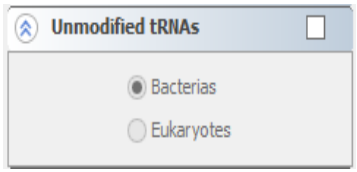
		CODONUSAGE1						
Gene Name	**Number of Codons**	**Weight**	**Maximize**	**Minimize**	**Harmonize**	**RSCU**	**CAI**	**Keep Rare**
yggN	240	0,456	0	1	0	1	0	1
yggL	119	0,027	0	0	1	0	1	0
ydfO	142	0,567	0	0	1	1	0	1
b1550	59	0,512	0	0	1	0	1	1
ribB	218	0,867	1	0	0	1	0	1
b3042	117	0,502	0	1	0	0	1	1
b2387	109	0,455	1	0	0	0	1	1
glk	322	0,427	0	0	1	1	0	1
tsx	295	0,523	0	0	1	0	1	1
yajI	200	0,994	0	1	0	1	0	0
b1009	267	0,298	1	0	0	1	0	0
b1010	129	0,966	1	0	0	1	0	1
rpsQ	85	0,572	0	1	0	1	0	1
rpmC	64	0,694	0	1	0	1	0	1
ydhD	116	0,373	0	1	0	1	0	1
sodB	194	0,045	0	0	1	0	1	0
ylcB	458	0,144	0	1	0	0	1	1
ylcC	111	0,168	1	0	0	1	0	1
yjeT	66	0,660	0	1	0	1	0	1
purA	433	0,727	1	0	0	1	0	0
recB	1181	0,404	1	0	0	1	0	1

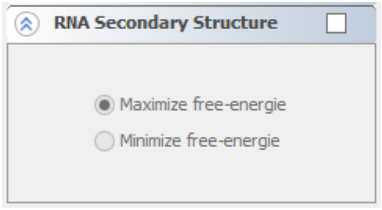
- 3 possible options
- Only one can be selected
- Custom option has a range of [1, 6] possible options



		HIDDENSTOPCODONS1									
Gene Name	**Number of Codons**	**Weight**	**Maximize**	**Minimize**	**Custom**	**1**	**2**	**3**	**4**	**5**	**6**
yggN	240	0,999	0	1	0	0	0	0	0	0	0
yggL	119	0,597	0	1	0	0	0	0	0	0	0
ydfO	142	0,45	0	0	1	0	1	0	0	0	0
b1550	59	0,569	0	0	1	0	0	0	0	1	0
ribB	218	0,663	0	1	0	0	0	0	0	0	0
b3042	117	0	0	0	1	0	0	0	0	0	1
b2387	109	0,623	1	0	0	0	0	0	0	0	0
glk	322	0,963	1	0	0	0	0	0	0	0	0
tsx	295	0,485	0	0	1	0	0	0	1	0	0
yajI	200	0,352	1	0	0	0	0	0	0	0	0
b1009	267	0,438	0	1	0	0	0	0	0	0	0
b1010	129	0,78	1	0	0	0	0	0	0	0	0
rpsQ	85	0,293	0	1	0	0	0	0	0	0	0
rpmC	64	0,091	0	1	0	0	0	0	0	0	0
ydhD	116	0,617	0	1	0	0	0	0	0	0	0
sodB	194	0,159	0	1	0	0	0	0	0	0	0
ylcB	458	0,514	0	0	1	0	0	0	0	0	1
ylcC	111	0,731	1	0	0	0	0	0	0	0	0
yjeT	66	0,509	0	0	1	0	0	0	0	1	0
purA	433	0,669	1	0	0	0	0	0	0	0	0
recB	1181	0,84	0	1	0	0	0	0	0	0	0

- 2 possible options in each plugin
- Only one can be selected





		UNMODIFIEDTRNAS1			RNASECONDARYSTRUCTURE1					
Gene Name	**Number of Codons**	**Weight**	**Bacterias**	**Eukaryotes**	**Weight**	**Maximize**	**Minimize**	**Elapsed Time**	**Initial Score**	**Final Score**
yggN	240	0,578	0	1	---	---	---	07:02:05	75,113	83,225
yggL	119	0,279	0	1	---	---	---	01:15:43	78,029	81,272
ydfO	142	0,270	0	1	---	---	---	01:01:09	76,071	82,556
b1550	59	0,093	1	0	---	---	---	00:18:46	76,483	84,640
ribB	218	0,359	1	0	---	---	---	04:52:29	75,679	85,840
b3042	117	0,282	1	0	---	---	---	00:42:08	75,053	85,889
b2387	109	0,913	1	0	---	---	---	01:06:11	77,523	82,660
glk	322	0,714	0	1	---	---	---	05:50:03	75,013	81,638
tsx	295	0,318	1	0	---	---	---	07:16:12	77,895	83,980
yajI	200	0,645	1	0	---	---	---	05:02:55	72,938	85,305
b1009	267	0,022	0	1	---	---	---	09:09:22	74,267	81,351
b1010	129	0,444	0	1	---	---	---	01:03:03	77,374	85,219
rpsQ	85	0,346	0	1	---	---	---	00:57:12	76,593	85,692
rpmC	64	0,359	1	0	---	---	---	00:14:16	74,382	89,510
ydhD	116	0,604	0	1	---	---	---	02:23:45	77,407	81,582
sodB	194	0,561	1	0	---	---	---	03:20:46	76,890	80,984
ylcB	458	0,903	0	1	---	---	---	31:32:23	72,858	77,531
ylcC	111	0,476	0	1	---	---	---	02:14:59	76,370	82,109
yjeT	66	0,696	1	0	---	---	---	00:28:16	75,884	84,115
purA	433	0,455	0	1	---	---	---	09:31:26	75,924	85,663
recB	1181	0,380	0	1	---	---	---	57:48:46	74,838	81,951

- Initial score represents the similarity percentage of the random genes generated regarding the original gene

- Final score represents the average similarity percentage of how similar a random gene become, regarding the original one, when applying to the set of redesign methods with the best parameters (represented by 1's)